Hardcore AI for Computer Games and Animation
SIGGRAPH 98 Course Notes

by

John David Funge

VISIT…

Abstract

Hardcore AI for Computer Games and Animation
SIGGRAPH 98 Course Notes

John David Funge
1998

Welcome to this tutorial on AI for Computer Games and Animation. These course notes consist of two parts:

Part I is a short overview that misses out lots of details.

Part II goes into all these details in great depth.

Biography

John Funge is a member of Intel's graphics research group. He received a BS in Mathematics from King's College London in 1990, a MS in Computer Science from Oxford University in 1991, and a PhD in Computer Science from the University of Toronto in 1998. It was during his time at Oxford that John became interested in computer graphics. He was commissioned by Channel 4 television to perform a preliminary study on a proposed computer game show. This made him acutely aware of the difficulties associated with developing *intelligent* characters. Therefore, for his PhD at the University of Toronto he successfully developed a new approach to high-level control of characters in games and animation. John is the author of several papers and has given numerous talks on his work, including a technical sketch at SIGGRAPH 97. His current research interests include computer animation, computer games, interval arithmetic and knowledge representation.

Hardcore AI for Computer Games and Animation
Siggraph Course Notes (Part I)

John Funge and Xiaoyuan Tu

Microcomputer Research Lab
Intel Corporation
{john_funge|xiaoyuan_tu}@ccm.sc.intel.com

Abstract

Recent work in behavioral animation has taken impressive steps toward autonomous, self-animating characters for use in production animation and computer games. It remains difficult, however, to *direct* autonomous characters to perform specific tasks. To address this problem, we explore the use of *cognitive models*. Cognitive models go beyond behavioral models in that they govern what a character knows, how that knowledge is acquired, and how it can be used to plan actions. To help build cognitive models, we have developed a cognitive modeling language (CML). Using CML , we can decompose cognitive modeling into first giving the character domain knowledge, and then specifying the required behavior. The character's domain knowledge is specified intuitively in terms of actions, their preconditions and their effects. To direct the character's behavior, the animator need only specify a behavior outline, or "sketch plan", and the character will automatically work out a sequence of actions that meets the specification. A distinguishing feature of CML is how we can use familiar control structures to focus the power of the reasoning engine onto tractable sub-tasks. This forms an important middle ground between regular logic programming and traditional imperative programming. Moreover, this middle ground allows many behaviors to be specified more naturally, more simply, more succinctly and at a much higher-level than would otherwise be possible. In addition, by using interval arithmetic to integrate sensing into our underlying theoretical framework, we enable characters to generate plans of action even when they find themselves in highly complex, dynamic virtual worlds. We demonstrate applications of our work to "intelligent" camera control, and behavior animation for characters situated in a prehistoric world and in a physics-based undersea world.

Keywords: Computer Animation, Knowledge, Sensing, Action, Reasoning, Behavioral Animation, Cognitive Modeling

1 Introduction

Modeling for computer animation addresses the challenge of automating a variety of difficult animation tasks. An early milestone was the combination of geometric models and inverse kinematics to simplify keyframing. Physical models for animating particles, rigid bodies, deformable solids, and fluids offer copious quantities of realistic motion through dynamic simulation. Biomechanical modeling employs simulated physics to automate the realistic animation of living things motivated by internal muscle actuators. Research in behavioral modeling is making progress towards self-animating characters that react appropriately to perceived environmental stimuli.

In this paper, we explore *cognitive modeling* for computer animation. Cognitive models go beyond behavioral models in that they govern what a character knows, how that knowledge is acquired, and how it can

be used to plan actions. Cognitive models are applicable to directing the new breed of highly autonomous, quasi-intelligent characters that are beginning to find use in animation and game production. Moreover, cognitive models can play subsidiary roles in controlling cinematography and lighting.

We decompose cognitive modeling into two related sub-tasks: *domain specification* and *behavior specification*. Domain specification involves giving a character knowledge about its world and how it can change. Behavior specification involves directing the character to behave in a desired way within its world. Like other advanced modeling tasks, both of these steps can be fraught with difficulty unless animators are given the right tools for the job. To this end, we develop a cognitive modeling language, CML.

CML rests on solid theoretical foundations laid by artificial intelligence (AI) researchers. This high-level language provides an intuitive way to give characters, and also cameras and lights, knowledge about their domain in terms of actions, their preconditions and their effects. We can also endow characters with a certain amount of "commonsense" within their domain and we can even leave out tiresome details from the specification of their behavior. The missing details are automatically filled in at run-time by a reasoning engine integral to the character which decides what must be done to achieve the specified behavior.

Traditional AI style planning certainly falls under the broad umbrella of this description, but the distinguishing features of CML are the intuitive way domain knowledge can be specified and how it affords an animator familiar control structures to focus the power of the reasoning engine. This forms an important middle ground between regular logic programming (as represented by Prolog) and traditional imperative programming (as typified by C). Moreover, this middle ground turns out to be crucial for cognitive modeling in animation and computer games. In one-off animation production, reducing development time is, within reason, more important than fast execution. The animator may therefore choose to rely more heavily on the reasoning engine. When run-time efficiency is also important our approach lends itself to an incremental style of development. We can quickly create a working prototype. If this prototype is too slow, it may be refined by including more and more detailed knowledge to narrow the focus of the reasoning engine.

2 Related Work

Badler [3] and the Thalmanns [13] have applied AI techniques [1] to produce inspiring results with animated humans. Tu and Terzopoulos [18] have taken impressive strides towards creating realistic, self-animating graphical characters through biomechanical modeling and the principles of behavioral animation introduced in the seminal work of Reynolds [16]. A criticism sometimes levelled at behavioral animation methods is that, robustness and efficiency notwithstanding, the behavior controllers are hard-wired into the code. Blumberg and Galyean [6] begin to address such concerns by introducing mechanisms that give the animator greater control over behavior and Blumberg's superb thesis considers interesting issues such as behavior learning [5]. While we share similar motivations, our work takes a different route. One of the features of our approach is that we investigate important higher-level cognitive abilities such as knowledge representation and planning.

The theoretical basis of our work is new to the graphics community and we consider some novel applications. We employ a formalism known as the situation calculus. The version we use is a recent product of the cognitive robotics community [12]. A noteworthy point of departure from existing work in cognitive robotics is that we render the situation calculus amenable to animation within highly dynamic virtual worlds by introducing interval valued fluents [9] to deal with sensing.

High-level camera control is particularly well suited to an approach like ours because there already exists a large body of widely accepted rules that we can draw upon [2]. This fact has also been exploited by two recent papers [10, 8] on the subject. This previous work uses a simple scripting language to implement hierarchical finite state machines for camera control.

3 Theoretical Background

The situation calculus is a well known formalism for describing changing worlds using sorted first-order logic. Mathematical logic is somewhat of a departure from the mathematical tools that have been used in previous work in computer graphics. In this section, we shall therefore go over some of the more salient points. Since the mathematical background is well-documented elsewhere (for example, [9, 12]), we only provide a cursory overview. We emphasize that from the user's point of view the underlying theory is *completely hidden*. In particular, a user is *not* required to type in axioms written in first-order mathematical logic. Instead, we have developed an intuitive interaction language that resembles natural language, but has a clear and precise mapping to the underlying formalism. In section 4, we give a complete example of how to use CML to build a cognitive model from the user's point of view.

3.1 Domain modeling

A *situation* is a "snapshot" of the state of the world. A domain-independent constant s_0 denotes the initial situation. Any property of the world that can change over time is known as a *fluent*. A fluent is a function, or relation, with a situation term as (by convention) its last argument. For example $\mathsf{Broken}(x, s)$ is a fluent that keeps track of whether an object x is broken in a situation s.

Primitive actions are the fundamental instrument of change in our ontology. The term "primitive" can sometimes be counter-intuitive and only serves to distinguish certain atomic actions from the "complex", compound actions that we will define in section 3.2. The situation s' resulting from doing action a in situation s is given by the distinguished function do, such that, $s' = \mathit{do}(a, s)$. The possibility of performing action a in situation s is denoted by a distinguished predicate $\mathit{Poss}(a, s)$. Sentences that specify what the state of the world must be before performing some action are known as *precondition axioms*. For example, it is possible to drop an object x in a situation s if and only if a character is holding it, $\mathit{Poss}(\mathit{drop}(x), s) \Leftrightarrow \mathsf{Holding}(x, s)$. In CML , this axiom can be expressed more intuitively without the need for logical connectives and the explicit situation argument. [1]

action *drop*(x) **possible when** Holding(x)

The convention in CML is that fluents to the left of the **when** keyword refer to the current situation. The effects of an action are given by *effect axioms*. They give necessary conditions for a fluent to take on a given value after performing an action. For example, the effect of dropping an object x is that the character is no longer holding the object in the resulting situation and vice versa for picking up an object. This is stated in CML as follows.

occurrence *drop*(x) **results in** !Holding(x)
occurrence *pickup*(x) **results in** Holding(x)

What comes as a surprise, is that, a naive translation of the above statements into the situation calculus does not give the expected results. In particular, there is a problem stating what does not change when an action is performed. That is, a character has to worry whether dropping a cup, for instance, results in a vase turning into a bird and flying about the room. For mindless animated characters, this can all be taken care of implicitly by the programmer's common sense. We need to give our thinking characters this same common sense. They have to be told that, unless they know better, they should assume things stay the same. In AI this is called the "frame problem" [14]. If characters in virtual worlds start thinking for themselves, then

[1] To promote readability all CML keywords will appear in bold type, actions (complex and primitive) will be italicized, and fluents will be underlined. We will also use various other predicates and functions that are not fluents. These will not be underlined and will have names to indicate their intended meaning.

they too will have to tackle the frame problem. Until recently, it is one of the main reasons why we have not seen approaches like ours used in computer animation or robotics.

Fortunately, the frame problem can be solved provided characters represent their knowledge in a certain way [15]. The idea is to assume that our effect axioms enumerate all the possible ways that the world can change. This closed world assumption provides the justification for replacing the effect axioms with *successor state* axioms. For example, the CML statements given above can now be effectively translated into the following successor state axiom that CML uses internally to represent how the character's world can change. It states that, provided the action is possible, then a character is holding an object x if and only if it just picked it up or it was holding it before and it did not just drop it, $\mathit{Poss}(a, s) \Rightarrow [\mathsf{Holding}(x, \mathit{do}(a, s)) \Leftrightarrow a = \mathit{pickup}(x) \vee (a \neq \mathit{drop}(x) \wedge \mathsf{Holding}(x, s))$.

3.1.1 Sensing

One of the limitations of the situation calculus, as we have presented it so far, is that we must always write down things that are true about the world. This works out fine for simple worlds as it is easy to place all the rules by which the world changes into the successor state axioms. Even in more complex worlds, fluents that represent the character we are controlling's internal state are, by definition, always true. Now imagine we have a simulated world that includes an elaborate thermodynamics model involving advection-diffusion equations. We would like to have a fluent temp that gives the temperature in the current situation for the character's immediate surroundings. What are we to do? Perhaps the initial situation could specify the correct temperature at the start? However, what about the temperature after a *setFireToHouse* action, or a *spillLiquidHelium* action, or even just twenty clock tick actions? We could write a successor state axiom that contains all the equations by which the simulated world's temperature changes. The character can then perform multiple forward simulations to know the precise effect of all its possible actions. This, however, is expensive, and even more so when we add other characters to the scene. With multiple characters, each character must perform a forward simulation for each of its possible actions, and then for each of the other character's possible actions and reactions, and then for each of its own subsequent actions and reactions, etc. Ignoring these concerns, imagine that we could have a character that can precisely know the ultimate effect of all its actions arbitrarily far off into the future. Such a character can see much further into its future than a human observer so it will not appear as "intelligent", but rather as "super-intelligent". We can think of an example of a falling tower of bricks where the character precomputes all the brick trajectories and realizes it is in no danger. To the human observer, who has no clue what path the bricks will follow, a character who happily stands around while bricks rain around it looks peculiar. Rather, the character should run for cover, or to some safe distance, based on its qualitative knowledge that nearby falling bricks are dangerous. In summary, we would like our characters to represent their uncertainty about some properties of the world until they sense them.

Half of the solution to the problem is to introduce *exogenous* actions (or events) that are generated by the environment and not the character. For example, we can introduce an action *setTemp* that is generated by the underlying simulator and simply sets the temperature to its current value. It is straightforward to modify the definition of complex actions, that we give in the next section, to include a check for any exogenous actions and, if necessary, include them in the sequence of actions that occur (see [9] for more details).

The other half of the problem is representing what the character knows about the temperature. Just because the temperature in the environment has changed does not mean the character should know about it until it performs a sensing action. In [17] sensing actions are referred to as *knowledge producing actions*. This is because they do not affect the world but only a character's knowledge of its world. The authors were able to represent a character's knowledge of its current situation by defining an epistemic fluent K to keep track of all the worlds a character thinks it might possibly be in. Unfortunately, the approach does not lend itself to easy implementation. The problems revolve around how to specify the initial situation. In general, if we have n relational fluents, whose value may be learned through sensing, then there will be 2^n initial

possible worlds that we potentially have to list out. Once we start using functional fluents, however, things get even worse: we cannot, by definition, list out the uncountably many possible worlds associated with not knowing the value of a fluent that takes on values in $\mathbb{R}$.

3.1.2 Interval-valued epistemic (IVE) fluents

The epistemic K-fluent allows us to express an agent's uncertainty about the value of a fluent in its world. Intervals arithmetic can also be used to express uncertainty about a quantity. Moreover, they allow us to do so in a way that circumvents the problem of how to use a finite representation for infinite quantities. It is, therefore, natural to ask whether we can also use intervals to replace the troublesome epistemic K-fluent. The answer, as we show in [9], is a resounding "yes". In particular, for each sensory fluent f, we introduce an interval-valued epistemic (IVE) fluent $\mathcal{I}_f$. The IVE fluent $\mathcal{I}_f$ is used to represent an agent's uncertainty about the value of f. Sensing now corresponds to making intervals narrower.

In our temperature example, we can introduce an IVE fluent, $\mathcal{I}_{\mathsf{temp}}$, that takes on values in $\mathcal{I}_{\mathbb{R}^{\star+}}$. Note that, $\mathcal{I}_{\mathbb{R}^{\star+}}$ denotes the set of pairs $\langle u, v\rangle$ such that $u, v \in \mathbb{R}^{\star+}$ and $u \leqslant v$. Intuitively, we can now use the interval $\mathcal{I}_{\mathsf{temp}}(\mathrm{s}_0) = \langle 10, 50\rangle$ to state that the temperature is initially known to be between 10 and 50 Kelvin. Now, as long as we have a bound on how fast the temperature changes, we can always write down true statements about the world. Moreover, we can always bound the rate of change. That is, in the worst case we can choose our rate of change as infinite so that, except after sensing, the character is completely ignorant of the temperature in the current situation $\mathcal{I}_{\mathsf{temp}}(s) = \langle 0, \infty\rangle$. Figure 1 depicts the more usual case when we do have a reasonable bound. The solid line is the actual temperature $\underline{\mathsf{temp}}$, and the shaded region is the interval that is guaranteed to bound the temperature. When the interval is less than a certain width we say that the character "knows" the property in question. We can then write precondition axioms based not only upon the state of the world, but also on the state of the character's knowledge of its world. For example, we can state that it is only possible to turn the heating up if the character knows it is too cold. If the character does not "know" the temperature (i.e. the interval $\mathcal{I}_{\mathsf{temp}}(s)$ is too wide) then the character can work out it needs to perform a sensing action. In [9] we prove many important equivalences, and theorems that allow us to justify using our IVE fluents to completely replace the troublesome K-fluent.

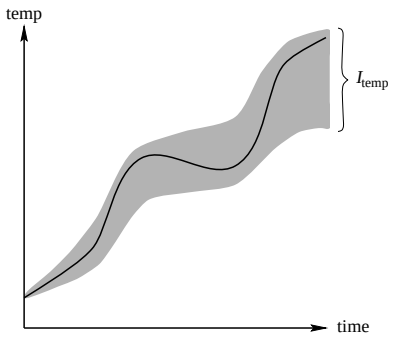


Figure 1: IVE fluents bound the actual fluents value

3.2 Behavior Modeling

Specifying behavior in CML capitalizes on our way of representing knowledge to include a novel approach to high-level control. It is based on the theory of *complex actions* from the situation calculus [12]. Any primitive action is also a complex action, and other complex actions can be built up using various control structures. As a familiar artifice to aid memorization, the control structure syntax of CML is deliberately chosen to resembles that of C.

Although the syntax may be similar to a conventional programming language, in terms of functionally CML is a strict superset. In particular, a behavior outline can be nondeterministic. By this, we do not mean that the behavior is random, but that we can cover multiple possibilities in one instruction. As we shall explain, this added freedom allows many behaviors to be specified more naturally, more simply, more succinctly and at a much higher-level than would otherwise be possible. The user can design characters based on behavior outlines, or "sketch plans". Using its background knowledge, the character can decide for itself how to fill in the necessary missing details.

The complete list of operators for defining complex actions is defined recursively and is given below. Together, they define the behavior specification language used for issuing advice to characters. The mathematical definitions for these operators are given in [12]. After each definition the equivalent CML syntax is given in square brackets.

(Primitive Action) If α is a primitive action then, provided the precondition axiom states it is possible, do the action [same except when the action is a variable when we need to use an explicit **do**];

(Sequence) $\alpha \fatsemi \beta$ means do action α, followed by action β [same except that in order to mimic C statements must end with a semi-colon];

(Test) $p?$ succeeds if p is true, otherwise it fails [**test**(<EXPRESSION>)];

(Nondeterministic choice of actions) $\alpha \mid \beta$ means do action α or action β [**choose** <ACTION> **or** <ACTION>];

(Conditionals) **if** p α **else** β **fi**, is just shorthand for $p? \fatsemi \alpha \mid (\neg p)? \fatsemi \beta$ [**if** (<EXPRESSION>) <ACTION> **else** <ACTION>];

(Non-deterministic iteration) $\alpha \star$, means do α zero or more times [**star** <ACTION>];

(Iteration) **while** p **do** α **od** is just shorthand for $p? \; \alpha \star$ [**while** (<EXPRESSION>) <ACTION>];

(Nondeterministic choice of arguments) $(\boldsymbol{\pi}\ x)\ \alpha$ means pick some argument x and perform the action $\alpha(x)$ [**pick**(<EXPRESSION>) <ACTION>];

(Procedures) **proc** $P(x_1, \ldots, x_n)$ α **end** declares a procedure that can be called as $P(x_1, \ldots, x_n)$ [**void** P(<ARGLIST>) <ACTION>].

As we mentioned earlier, the purpose of CML is not simply that it be used for conventional planning, but to illustrate its power consider that the following complex action implements a depth-first planner. The CML version is given alongside.[2]

proc *planner*(n)
  goal? |
  [$(n > 0)? \fatsemi$
    $(\boldsymbol{\pi}\ a)$(primitiveAction$(a)? \fatsemi a) \fatsemi$
    planner$(n \Leftrightarrow 1)$]
end

proc *planner*(n) {
  choose test(goal);
  or {
    test$(n > 0)$;
    pick(a) {
      primitiveAction(a);
      do(a);
    }
  }
  planner$(n \Leftrightarrow 1)$;
}

[2]For the remainder of this paper we will use CML syntax.

Assuming we define what the primitive actions are, and the goal, then this procedure will perform a depth-first search for plans of length less than n. We have written a Java applet, complete with documentation, that is available on the World Wide Web to further assist the interested reader in mastering this novel language [11].

The following maze example is not meant to be a serious application. It is a simple, short tutorial designed to explain how an animator would use CML.

4 Simple maze example

A maze is defined as a finite grid with some occupied cells. We say that a cell is Free if it is in the grid and not occupied. A function adjacent returns the cell that is adjacent to another in a particular direction. Figure 2 shows a simple maze, some examples of the associated definitions, and the values of the fluents in the current situation. There are two fluents, position denotes which cell contains the character in the current situation, and visited denotes the cells the character has previously been to.

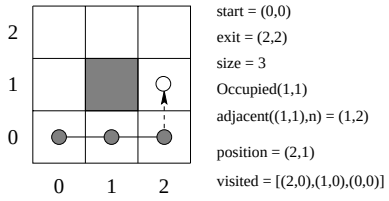


Figure 2: A simple maze

The single action in this example is a *move* action that takes one of four compass directions as a parameter. It is possible to move in some direction d, provided the cell we are moving to is free, and has not been visited before.

action *move*(d) **possible when** c = adjacent(position,d) && Free(c) && !member(c,visited);

Figure 3 shows the possible directions a character can move when in two different situations.

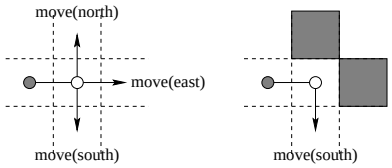


Figure 3: Possible directions to move

A fluent is completely specified by its initial value, and its successor state axiom. For example, the initial position is given as the start point of the maze, and the effect of moving to a new cell is to update the position accordingly.

initially position = start;
occurrence *move*(d) **results in** position = adjacent(p_{old},d) when position = p_{old};

The fluent visited is called a *defined* fluent because it is defined (recursively) in terms of the previous position, and the previous visited cells. Therefore, its value changes implicitly as the position changes. The user must be careful to avoid any circular definitions when using defined fluents. A defined fluent is

indicated with a ":=". Just as with regular fluents, anything to the left of a **when** refers to the previous situation.[3]

initially visited := [];
visited := [$p_{old} \mid v_{old}$] **when** position = p_{old} && visited = v_{old};

The behavior we are interested in specifying in this example is that of navigating a maze. The power of CML allows us to express this fact directly as follows.

while (position != exit)
  pick(d) *move*(d);

Just like a regular while loop, the above program expands out into a sequence of actions. Unlike a regular while loop it expands out not into one particular sequence of actions, but into *all possible* sequences of actions. A possible sequence of actions is defined by the precondition axioms that we previously stated, and the exit condition of the loop. Therefore, any free path through the maze, that does not backtrack, and ends at the exit position meets the behavior specification. This is what we mean by a nondeterministic behavior specification language. Nothing "random" is happening, we can simply specify a large number of possibilities all at once. Searching the specification for valid action sequences is the job of an underlying reasoning engine. Figure 4 depicts all the behaviors that meet the above specification for the simple maze we defined earlier. [4]

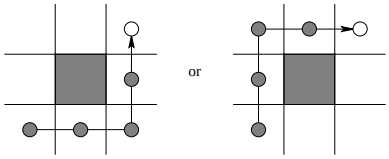


Figure 4: Valid behaviors

Although we disallow backtracking in the final path through the maze, the reasoning engine may use backtracking when it is searching for valid paths. In the majority of cases, the reasoning engine can use depth-first search to find a path through a given maze in a few seconds. To speed things up, we can easily start to reduce some of the nondeterminism by specifying a "best-first" search strategy. In this approach, we will not leave it up to the character to decide how to search the possible paths, but constrain it to first investigate paths that head toward the exit. This requires extra lines of code but could result in faster execution.

For example, suppose we add an action *goodMove*, such that it is possible to move in a direction d if it is possible to "*move*" to the cell in that direction, and the cell is closer to the goal than we are now:

action *goodMove*(d) **possible when possible**(*move*(d)) && Closer(exit,d,position);

Now we can rewrite our high-level controller as one that prefers to move toward the exit position whenever possible.

while (position != exit)
  choose pick (d) *goodMove*(d);
  or pick (d) *move*(d);

[3] Here we are using Prolog list notation.

[4] Mathematically we have that the final situation $s' = do(move(n), do(move(n), do(move(e), do(move(e), S_0)))) \vee s' = do(move(e), do(move(e), do(move(n), do(move(n), S_0))))$.

At the extreme, there is nothing to prevent us from coding in a simple deterministic strategy such as the “left-hand” rule. The important point is that our approach does not rule out any of the algorithms one might consider when writing the same program in C. Rather, it opens up new possibilities for very high-level specifications of behavior.

5 Cinematography

We have positioned our work as dealing with cognitive modeling. At first, it might seem strange to be advocating building a cognitive model for a camera. We soon realize, however, that it is the knowledge of the cameraperson, and the director, who control the camera that we want to capture with our cognitive model. In effect, we want to treat all the components of a scene, be they lights, cameras, or characters as ”actors”. Moreover, CML is ideally suited to realizing this approach.

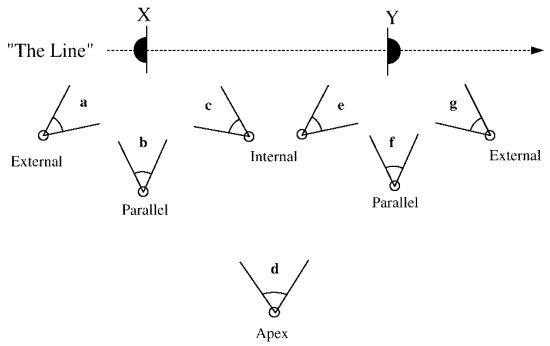


Figure 5: Common camera placements

To appreciate what follows, the reader may benefit from a rudimentary knowledge of cinematography (see 5). The exposition given in section 2.3, “Principles of cinematography”, of [10] is an excellent starting point. In [10], the authors discuss one particular formula for filming two characters talking to one another. The idea is to flip between “external” shots of each character, focusing on the character doing the talking. To break up the monotony, the shots are interspersed with reaction shots of the other character. In [10], the formula is encoded as a finite state machine. We will show how elegantly we can capture the formula using the behavior specification facilities of CML. Firstly, however, we need to specify the domain. In order to be as concise as possible, we shall concentrate on explaining the important aspects of the specification, any missing details can be found in [9].

5.1 Camera domain

We shall be assuming that the motion of all other objects in the scene has been computed. Our task is to decide, for each frame, the vantage point from which it is to be rendered. The fluent frame keeps track of the current frame number, and a *tick* action causes it to be incremented by one. The precomputed scene is represented as a lookup function, scene, which for each object, and each frame, completely specifies the position, orientation, and shape.

The most common camera placements used in cinematography will be modeled in our formalization as primitive actions. In [10], these actions are referred to as “camera modules”. This is a good example where the term “primitive” is misleading. As described in [4], low-level camera placement is a complex and challenging task in its own right. For the purposes of our exposition here we shall make some simplifications. More realistic equations can easily be substituted, but the principals remain the same. For now, we specify the camera with two fluents lookFrom, and lookAt. Let us assume that up remains constant. In addition, we

also make the simplifying assumption that the viewing frustrum is fixed. Despite our simplifications, we still have a great deal of flexibility in our specifications. We will now give examples of effect axioms for some of the primitive actions in our ontology.

The *fixed* action is used to explicitly specify a particular camera configuration. We can, for example, use it to provide an overview shot of the scene:

occurrence *fixed*(e,c) **results in** lookFrom = e && lookAt = c;

A more complicated action is *external*. It takes two arguments, character A, and character B and places the camera so that A is seen over the shoulder of B. One effect of this action, therefore, is that the camera is looking at character A:

occurrence *external*(A,B) **results in** lookAt = p **when** scene(A(upperbody,centroid)) = p;

The other effect is that the camera is located above character B's shoulder. This might be accomplished with an effect axiom such as:

occurrence *external*(A,B) **results in** lookFrom = $p + k_2 *$ up $+ k_3 *$ normalize$(p \Leftrightarrow c)$ **when**
scene(B(shoulder,centroid)) = p && scene(A(upperbody,centroid)) = c;

where k_2 and k_3 are some suitable constants.

There are many other possible camera placement actions. Some of them are listed in [10], others may be found in [2].

The remaining fluents are concerned with more esoteric aspects of the scene, but some of their effect axioms are mundane and so we shall only explain them in English. For example, the fluent Talking(A,B) (meaning A is talking to B) becomes true after a *startTalk*(A,B) action, and false after a *stopTalking*(A,B) action. Since we are currently only concerning ourselves with camera placement it is the responsibility of the application that is generating the scene descriptions to produce the start and stop talking actions. more interesting fluent is silenceCount, it keeps count of how long it has been since a character spoke.

occurrence *tick* **results in** silenceCount = $n \Leftrightarrow 1$ **when** silenceCount = n && !exists(A, B) Talking(A,B);
occurrence *stopTalk*(A,B) **results in** silenceCount = k_a;
occurrence *setCount* **results in** silenceCount = k_a;

Note that, k_a is a constant ($k_a = 10$ in [10]), such that after k_a ticks of no-one speaking the counter will be negative. A similar fluent filmCount keeps track of how long the camera has been pointing at the same character:

occurrence *setCount* || *external*(A,B) **results in** filmCount = k_b **when** Talking(A,B);
occurrence *setCount* || *external*(A,B) **results in** filmCount = k_c **when** !Talking(A,B);
occurrence *tick* **results in** filmCount = $n \Leftrightarrow 1$ **when** filmCount = n;

k_b and k_c are constants ($k_b = 30$ and $k_c = 15$ in [10]) that state how long we can stay with the same shot before the counter becomes negative. Note that, the constant for the case of looking at a non-speaking character is lower. We will keep track of which constant we are using with the fluent tooLong.

For convenience, we now introduce two defined fluents that express when a shot has become boring because it has gone on too long, and when a shot has not gone on long enough. We need the notion of a minimum time for each shot to avoid instability that would result in flitting between one shot and another too quickly.

Boring := filmCount < 0;
TooFast := tooLong - $k_s \leqslant$ filmCount;

Finally, we introduce a fluent Filming to keep track of who the camera is pointing at.

Until now, we have not mentioned any preconditions for our actions. The reader may assume that, unless stated otherwise, all actions are always possible. In contrast, the precondition axiom for the *external* camera action states that we only want to be able to point the camera at character A, if we are already filming A, and it has not got boring yet; or we not filming A, and A is talking, and we have stayed with the current shot long enough:

action *external*(A,B) **possible when** (!Boring && Filming(A)) || (Talking(A,B) && !Filming(A) && !TooFast;

We are now in a position to define the controller that will move the camera to look at the character doing the talking, with occasional respites to focus on the other character's reactions:

setCount;
while ($0 <$ silenceCount) {
  pick(A,B) *external*(A,B);
  tick;
}

As in the maze solving example, this specification makes heavy use of the ability to nondeterministically choose arguments. The reader might like to contrast this definition with the encoding given in [10] to achieve the same result (see appendix).

6 Behavioral Animation

We now turn our attention behavioral animation, which is the other main application that we have discovered for our work. The first example we consider is prehistoric world, and the second is an undersea world. The undersea world is differentiated by the complexity of the underlying model.

6.1 Prehistoric world

In our prehistoric world we have a Tyrannosauras Rex (T-Rex) and some Velociprators (Raptors). The motion is generated by some simplified physics and a lot of inverse kinematics. The main non-aesthetic advantage the system has is that it is real-time on a Pentium II with an Evans and Sutherland RealImage 3D Graphics card. Our CML specifications were compiled into Prolog using our online applet [11]. The resulting Prolog code was then compiled into the underlying system using Quintus Prolog's ability to link with Visual C++. Unfortunately, performance was too adversely affected so we wrote our own reasoning engine, from scratch, in Visual C++. Performance is real-time on average, but can be slightly jittery when the reasoning engine takes longer than usual to decide on some suitable behavior.

So far we have made two animations using the dinosaurs. The first one was to show our approach applied to camera control, and the second has to do with territorial behavior. Although some of the camera angles are slightly different, the camera animation uses essentially the same CML code as the example given in section 5. The action consists of a T-Rex and a Raptor having a "conversation". Some frames from an animation called "Cinemasauras" are shown in the color plates at the end of the paper.

The territorial T-Rex animation was inspired by the work described in [7] in which a human tries using a virtual reality simulator to herd reactive characters. Our challenge was to have the T-Rex eject some Raptors from its territory. The Raptors were defined to be afraid of the T-Rex (especially if it roared) and so they would try and run in the opposite direction if it got too close. The T-Rex, therefore, had to try and get in behind the Raptors and frighten them toward a pass, while being careful not to frighten the raptors that were already going in the right direction. The task was made particularly hard because the T-Rex is slower and less maneuverable than the Raptors and so it needs to be smarter in deciding where to go. Programming

this as a reactive system would be non-trivial. The CML code we used was similar to the planner listed in section 3.2 except that it was written to search breadth-first. To generate the animation all we essentially had to do was to define the fluent goal in terms of the fluent that tracks the number of Raptors heading in the right direction, numRightDir. In particular, goal is true if there are k more raptors heading in the right direction than there are currently.

goal := numRightDir = n && $n_0 + k <= n$ **when initially** numRightDir = n_0

To speed things up we also defined a fluent badSituation which we can use to prune the search space. For example, if the T-Rex just roared, and no raptors changed direction, then we are in a bad situation:

badSituation **after** *roar* := noInWrongDir = n && $m = n$ **when** noInWrongDir = n_0 && $n_0 + k <= n$

If the T-rex cannot find a sequence of actions that it believes will get k raptors heading in the right direction, as long as it makes some partial progress, it will settle for the best it could come up with. If it cannot find a sequence of actions that result in even partial progress (for example when the errant raptors are too far away) it looks for a simple alternative plan to just move closer to nearby Raptors that are heading in the wrong direction. The information computed to find a primary plan can still be used to avoid frightening any raptors unnecessarily as it plans a path to move toward the errant ones.

6.2 Undersea world

In our undersea world we bring to life some mythical creatures, namely "merpeople". The undersea world is physics-based. The high-level intentions of a merperson get filtered down into detailed muscle actions which cause reaction forces on the virtual water. This makes it hard for a merperson to reason about its world as it is difficult to predict the ultimate effect of their actions. A low-level reactive behavior system helps to some extent by providing a buffer between the reasoning engine and the environment. Thus at the higher level we need only consider actions such as "go left", "go to a specific position", etc. and the reactive system will take care of translating these commands down into the required detailed muscle actions. Even so, without the ability to perform precise multiple forward simulations the exact position that a merperson will end up, after executing a plan of action, is hard for the reasoning engine to predict. A typical solution would be to re-initialize the reasoning engine every time it is called, but this makes it difficult to pursue long term goals as we are throwing out all the characters knowledge instead of just the knowledge that is out of date.

The solution is for the characters to use the IVE fluents that we described in section 3.1.2 to represent positions. After sensing the positions of all the characters that are visible are known. The merperson can then use this knowledge to replan its course of action, possibly according to some long-term strategy. Regular fluents are used to model the merperson's internal state, such as its goal position, fear level, etc.

6.2.1 Reasoning and Reactive system

The relationship between the user, the reasoning system and the reactive system is depicted in figure 6.

The reactive system provides us with virtual creatures that are fully functional autonomous agents. On its own it provides an operational behavior system. The system provides: A *graphical display model* that captures the form and appearance of our characters; A *biomechanical model* that captures the physical and anatomical structure of the character's body, including its muscle actuators, and simulates its deformation and physical dynamics; A *behavioral control model* that is responsible for motor control, perception control and behavior control of the character. Although this control model is capable of generating some high-level behaviors, we need only the low-level behavior capabilities.

Most of the relevant details of the reactive behavior system are given in [18]. The reactive system also acts as a fail-safe, should the reasoning system temporarily fall through. The reactive layer can thus be used

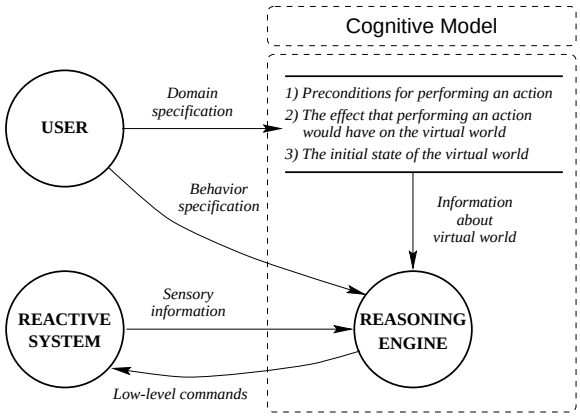


Figure 6: Interaction between cognitive model, user and low-level reactive behavior system.

to avoid the character doing anything "stupid" in the event that it cannot decide on anything "intelligent". Behaviors such as "continue in the same direction", "avoiding collisions" are examples of typical default reactive system behaviors.

The complete listing of the code (using an older version of CML) that we used to generate the animations is available in appendix F of [9]. The reasoning system runs on a Sun UltraSPARC, and the the reactive system runs simultaneously on an SGI Indigo 2 Extreme. On its own the reactive system manages about 3 frames per second and this slows to about 1 frame per second with reasoning. A large part of the extra overhead is accounted for by reading and writing files that the reactive and reasoning system use to communicate.

6.2.2 Undersea Animations

The undersea animations revolve around pursuit and evasion behaviors. The sharks try to eat the merpeople and the merpeople try to use the superior reasoning abilities we give them to avoid such a fate. For the most part, the sharks are instructed to chase merpeople they see. If they cannot see any, they go to where they last saw one. If all else fails they start to search systematically. The "Undersea Animation" color plates at the end show selected frames from two particular animations.

The first animations we produced were to verify that the shark could easily catch a merman swimming in open water. The shark is larger and swims faster, so it has no trouble catching its prey. Next, we introduced some obstacles. Now when the merman is in trouble it can come up with short term plans to take advantage of undersea rocks to frequently evade capture. It can hide behind the rocks and hug them closely so that the shark has difficulty seeing or reaching it. We were able to use the control structures of CML to encode a great deal of heuristic knowledge. For example, consider the problem of trying to come up with a plan to hide from a predator. A traditional planning approach will be able to perform a search of various paths according to criteria such as whether it uses hidden positions, whether it is far from a predator, etc. Unfortunately, this kind of planning is expensive and therefore can not be done over long distances. By using the control structures of CML , we can encode various heuristic knowledge to help overcome this limitation. For example, we can specify a procedure that encodes the following heuristic: if the current position is good enough then stay where you are; otherwise search the area around you (the expensive planning part); otherwise check out the obstacles (hidden positions are more likely near obstacles); if all else fails panic and go in a random direction. With a suitable precondition for *pickGoal*, that prevents the merperson selecting a goal until it meets a certain minimum criteria, the following CML procedure implements the above heuristic for character i.

void evade(i) $\{$

choose *testCurrPosn*(i)
or *search*(i)
or *testObstacles*(i)
or *panic*(i)
}

In turn, the above procedure can be part of a larger program that causes a merperson to hide from sharks while, say, trying to visit the other rocks in the scene whenever it is safe to do so. Of course, planning is not always a necessary, appropriate, or a possible, way to generate every aspect of an animation. This is especially so if an animator has something highly specific in mind. In this case, it is important to remember that CML has the full-range of control structures that we are used to in any regular programming language. We used these control structures to make the animation "The Great Escape". This was done by simply instructing the merman to avoid being eaten, and whenever it appears reasonably safe to do so, to make a break for a particular rock in the scene. The particular rock that we want to get the merman to go to has the property that it contains a narrow crack through which the merman can pass but through which the shark can not. What we wanted was an animation in which the merman eventually gets to the special rock with the shark in hot pursuit. The merman's *evade* procedure should then swing into action, hopefully causing it to evade capture by slipping through the crack. Although we do not know exactly when, or how, we have a mechanism to heavily stack the deck toward getting what we want. In our case, we got what we wanted first time but if it remained elusive we can carry on using CML , just like a regular programming language, to constrain what happens all the way down to scripting an entire sequence if we have to.

Finally, as an extension to behavior animation our approach inherits the ability to linearly scale a single character. That is, once we have developed a cognitive model for one character, we can reuse the model to create multiple characters. Each character will behave autonomously, according to their own unique perspective of their virtual world.

7 Conclusion

There is a large scope for future work. We could integrate a mechanism to learn reactive rules that mimic the behavior observed from the reasoning engine. Other issues arise in the user interface. As it stands CML is a good choice as the underlying representation a developer might want to use to build a cognitive model. An animator, or other users, might prefer a graphical user interface as a front-end. In order to be easy to use we might limit the interaction to supplying parameters to predefined models, or perhaps we could use a a visual programming metaphor to specify the complex actions.

In summary, CML always gives us an intuitive way to give a character knowledge about its world in terms of actions, their preconditions and their effects. When we have a high-level description of the ultimate effect of the behavior we want from a character, then CML gives us a way to automatically search for suitable action sequences. When we have a specific action sequence in mind, there may be no point to have CML search for one. In this case, we can use CML more like a regular programming language, to express precisely how we want the character to behave. We can even use a combination of these two extremes, and the whole gamut inbetween, to build different parts of one cognitive model. It is this combination of convenience and automation that makes CML such a potentially important tool in the arsenal of tomorrow's animators and game developers.

8 Acknowledgements

We would like to thank Eugene Fiume for originally suggesting the application of CML to cinematography, and Angel Studios for developing the low-level dinosaur API.

References

[1] J. Allen, J. Hendler, and A. Tate, editors. *Readings in Planning*. Morgan Kaufmann, 1990.

[2] D. Arijon. *Grammar of the Film Language*. Communication Arts Books, Hastings House, Publishers, New York, 1976.

[3] N. I. Badler, C. Phillips, and D. Zeltzer. *Simulating Humans*. Oxford University Press, 1993.

[4] J. Blinn. Where am i? what am i looking at? In *IEEE Computer Graphics and Applications*, pages 75–81, 1988.

[5] B. Blumberg. *Old Tricks, New Dogs: Ethology and Interactive Creatures*. PhD thesis, MIT Media Lab, MIT, Boston, USA, 1996.

[6] B. M. Blumberg and T. A. Galyean. Multi-level direction of autonomous creatures for real-time environments. In R. Cook, editor, *Proceedings of SIGGRAPH '95*, pages 47–54. ACM SIGGRAPH, ACM Press, Aug. 1995.

[7] D. Brogan, R. A. Metoyer, and J. K. Hodgins. Dynamically simulated characters in virtual environments. In *Animation Sketch, Siggraph '97*, 1997.

[8] D. B. Christianson, S. E. Anderson, L. He, D. H. Salesin, D. Weld, and M. F. Cohen. Declarative camera control for automatic cinematography. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI–96)*, Menlo Park, CA., 1996. AAAI Press.

[9] —, — PhD thesis, 1997.—

[10] L. He, M. F. Cohen, and D. Salesin. The virtual cinematographer: A paradigm for automatic real-time camera control and directing. In H. Rushmeier, editor, *Proceedings of SIGGRAPH '96*, Aug. 1996.

[11] — *CML Compiler Applet*, 1997.

[12] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.

[13] N. Magnenat-Thalmann. *Computer Animation: Theory and Practice*. Springer-Verlag, second edition edition, 1990.

[14] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, 1969.

[15] R. Reiter. The frame problem in the situation calculus. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation*, pages 359–380,418–420. Academic Press, 1991.

[16] C. W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. In M. C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 25–34, July 1987.

[17] R. Scherl and H. Levesque. The frame problem and knowledge-producing actions. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI–93)*, 1993. AAAI Press.

[18] X. Tu and D. Terzopoulos. Artificial fishes: Physics, locomotion, perception, behavior. In A. Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, pages 43–50. July 1994.

A Camera Code from [10]

```
DEFINE_IDIOM_IN_ACTION(2Talk)
     WHEN ( talking(A, B) )
          DO ( GOTO (1); )
     WHEN ( talking(B, A) )
          DO ( GOTO (2); )
END_IDIOM_IN_ACTION

DEFINE_STATE_ACTIONS(COMMON)
     WHEN (T < 10)
          DO ( STAY; )
     WHEN (!talking(A, B) && !talking(B, A))
          DO ( RETURN; )
END_STATE_ACTIONS

DEFINE_STATE_ACTIONS(1)
     WHEN ( talking(B, A) )
          DO ( GOTO (2); )
     WHEN ( T > 30 )
          DO ( GOTO (4); )
END_STATE_ACTIONS

DEFINE_STATE_ACTIONS(2)
     WHEN ( talking(A, B) )
          DO ( GOTO (1); )
     WHEN ( T > 30 )
          DO ( GOTO (3); )
END_STATE_ACTIONS

DEFINE_STATE_ACTIONS(3)
     WHEN ( talking(A, B) )
          DO ( GOTO (1); )
     WHEN ( talking(B, A) && T > 15 )
          DO ( GOTO (2); )
END_STATE_ACTIONS

DEFINE_STATE_ACTIONS(4)
     WHEN ( talking(B, A) )
          DO ( GOTO (2); )
     WHEN ( talking(A, B) && T > 15 )
          DO ( GOTO (1); )
END_STATE_ACTIONS.
```

Hardcore AI for Computer Games and Animation

SIGGRAPH 98 Course Notes (Part II)

by

John David Funge

Abstract

Hardcore AI for Computer Games and Animation
SIGGRAPH 98 Course Notes (Part II)

John David Funge
1998

For applications in computer game development and character animation, recent work in behavioral animation has taken impressive steps toward autonomous, self-animating characters. It remains difficult, however, to *direct* autonomous characters to perform specific tasks. We propose a new approach to high-level control in which the user gives the character a behavior outline, or "sketch plan". The behavior outline specification language has syntax deliberately chosen to resemble that of a conventional imperative programming language. In terms of functionality, however, it is a strict *superset*. In particular, a behavior outline need not be deterministic. This added freedom allows many behaviors to be specified more naturally, more simply, more succinctly and at a much higher-level than would otherwise be possible. The character has complete autonomy to decide on how to fill in the necessary missing details.

The success of our approach rests heavily on our use of a rigorous logical language, known as the *situation calculus*. The situation calculus is well-known, simple and intuitive to understand. The basic idea is that a character views its world as a sequence of "snapshots" known as situations. An understanding of how the world can change from one situation to another can then be given to the character by describing what the effect of performing each given action would be. The character can use this knowledge to keep track of its world and to work out which actions to do next in order to attain its goals. The version of the situation calculus we use incorporates a new approach to representing epistemic fluents. The approach is based on interval arithmetic and addresses a number of difficulties in implementing previous approaches.

Contents

1 Introduction **1**

1.1 Previous Models . . . 1
1.2 Cognitive models . . . 2
1.3 Aims . . . 3
1.4 Challenges . . . 3
1.5 Methodology . . . 5
1.6 Overview . . . 6

2 Background **9**

2.1 Kinematics . . . 9
2.1.1 Geometric Constraints . . . 9
2.1.2 Rigid Body Motion . . . 9
2.1.3 Separating Out Rigid Body Motion . . . 10
2.1.4 Articulated Figures . . . 10
Forward Kinematics . . . 11
Inverse Kinematics . . . 12
2.2 Kinematic Control . . . 12
2.2.1 Key-framing . . . 12
2.2.2 Procedural Control . . . 13
2.3 Noninterpenetration . . . 13
2.3.1 Collision Detection . . . 13
2.3.2 Collision Resolution and Resting Contact . . . 14
2.4 Dynamics . . . 14
2.4.1 Physics for Deformable Bodies . . . 15
2.4.2 Physics for Articulated Rigid Bodies . . . 15
Lagrange's Equation . . . 15
Newton-Euler Formulation . . . 15
2.4.3 Forward Dynamics . . . 16
2.4.4 Inverse Dynamics . . . 16
2.4.5 Additional Geometric Constraints . . . 17
2.5 Realistic Control . . . 17
2.5.1 State Space . . . 17
2.5.2 Output Vector . . . 17
2.5.3 Input Vector . . . 17

2.5.4 Control Function . . . 18
Hand-crafted Controllers . . . 18
Control Through Optimization . . . 19
Objective Based Control . . . 19
2.5.5 Synthesizing a Control Function . . . 20
2.6 High-Level Requirements . . . 21
2.7 Our Work . . . 22

3 Theoretical Basis **25**
3.1 Sorts . . . 25
3.2 Fluents . . . 26
3.3 The Qualification Problem . . . 26
3.4 Effect Axioms . . . 27
3.5 The Frame Problem . . . 27
3.5.1 The Ramification Problem . . . 28
3.6 Complex Actions . . . 28
3.7 Exogenous Actions . . . 30
3.8 Knowledge producing actions . . . 30
3.8.1 An epistemic fluent . . . 31
3.8.2 Sensing . . . 31
3.8.3 Discussion . . . 32
Implementation . . . 32
Real numbers . . . 33
3.9 Interval arithmetic . . . 33
3.10 Interval-valued fluents . . . 34
3.11 Correctness . . . 36
3.12 Operators for interval arithmetic . . . 38
3.13 Knowledge of terms . . . 39
3.14 Usefulness . . . 40
3.15 Inaccurate Sensors . . . 43
3.16 Sensing Changing Values . . . 44
3.17 Extensions . . . 45

4 Kinematic Applications **47**
4.1 Methodology . . . 47
4.2 Example . . . 47
4.3 Utilizing Non-determinism . . . 48
4.4 Another example . . . 49
4.4.1 Implementation . . . 52
4.4.2 Intelligent Flocks . . . 54
4.5 Camera Control . . . 54
4.5.1 Axioms . . . 55
4.5.2 Complex actions . . . 56

5 Physics-based Applications **61**
5.1 Reactive System . . . 61
5.2 Reasoning System . . . 62
5.3 Background Domain Knowledge . . . 62
5.4 Phenomenology . . . 63
5.4.1 Incorporating Perception . . . 64
Rolling forward . . . 64
Sensing . . . 65
5.4.2 Exogenous actions . . . 65
5.5 Advice through "Sketch Plans" . . . 65
5.6 Implementation . . . 66
5.7 Correctness . . . 68
5.7.1 Visibility Testing . . . 70
5.8 Reactive System Implementation . . . 73
5.8.1 Appearance . . . 73
3D Geometric Models . . . 74
Texture Mapping . . . 75
5.8.2 Locomotion . . . 75
Deformable models . . . 76
5.8.3 Articulated Figures . . . 78
5.8.4 Locomotion Learning . . . 78
5.8.5 Perception . . . 78
5.8.6 Behavior . . . 78
Collision Avoidance . . . 79
5.9 Animation Results . . . 80
5.9.1 Nowhere to Hide . . . 82
5.9.2 The Great Escape . . . 82
5.9.3 Pet Protection . . . 84
5.9.4 General Mêlée . . . 87

6 Conclusion **89**
6.1 Summary . . . 89
6.2 Future Work . . . 89
6.3 Conclusion . . . 91

A Scenes **93**

B Homogeneous Transformations **95**

C Control Theory **97**

D Complex Actions **99**

E Implementation **101**
E.1 Overall structure . . . 102

E.2 Sequences . . . 103
E.3 Tests . . . 104
E.4 Conditionals . . . 105
E.5 Nondeterministic iteration . . . 107
E.6 While loops . . . 108
E.7 Nondeterministic choice of action . . . 109
E.8 Nondeterministic choice of arguments . . . 109
E.9 Procedures . . . 110
E.10 Miscellaneous features . . . 112

F Code for Physics-based Example **115**
F.1 Procedures . . . 115
F.2 Pre-condition axioms . . . 116
F.3 Successor-state axioms . . . 116

Bibliography **120**

List of Figures

1.1 Shifting the burden of the work. . . . 2
1.2 Many possible worlds. . . . 5
1.3 Interaction between CDW, the animator and the low-level reactive behavior system. . . . 6

2.1 Kinematics . . . 10
2.2 Joint and Link Parameters . . . 11
2.3 "Muscle" represented as a spring and damper . . . 18
2.4 Design Space . . . 22

3.1 After sensing, only worlds where the light is on are possible. . . . 31

4.1 Some frames from a simple airplane animation. . . . 49
4.2 A simple maze. . . . 49
4.3 Visited cells. . . . 50
4.4 Choice of possibilities for a next cell to move to. . . . 51
4.5 Just one possibility for a next cell to move to. . . . 51
4.6 Updating maze fluents. . . . 51
4.7 A path through a maze. . . . 53
4.8 Camera placement is specified relative to "the Line" (Adapted from figure 1 of [He96]). . . . 55

5.1 Cell A and B are "completely visible" from one another . . . 71
5.2 Cell A and B are "completely occluded" from one another . . . 71
5.3 Cell A and B are "partially occluded" from one another . . . 72
5.4 Visibility testing near an obstacle. . . . 73
5.5 The geometric model. . . . 74
5.6 Coupling the geometric and dynamic model. . . . 74
5.7 Texture mapped face. . . . 75
5.8 A merperson swimming. . . . 76
5.9 The dynamic model. . . . 76
5.10 The repulsive potential. . . . 80
5.11 The attractive potential. . . . 81
5.12 The repulsive and attractive potential fields. . . . 81
5.13 Nowhere to Hide (part I) . . . 82
5.14 Nowhere to Hide (part II) . . . 83
5.15 The Great Escape (part I) . . . 83

5.16 The Great Escape (part II) 84
5.17 The Great Escape (part III) 85
5.18 The Great Escape (part IV) 85
5.19 Pet Protection (part I) 86
5.20 Pet Protection (part II) 86
5.21 Pet Protection (part III) 87
5.22 General Mêlée (part I) 88
5.23 General Mêlée (part II) 88

Chapter 1

Introduction

Computer animation is concerned with producing sequences of images (or frames) that when displayed in order, at sufficiently high speed, give the illusion of recognizable components of the image moving in recognizable ways. It is possible to place requirements on computer animations such as "objects should look realistic", or "objects should move realistically". The traditional approach to meeting these requirements was to employ skilled artists and animators. The talents of the most highly skilled human animators may still equal or surpass what might be attainable by computers. However, not everyone who wants to, or needs to, produce good quality animations has the time, patience, ability or money to do so. Moreover, for certain types of applications, such as computer games, human involvement in run-time satisfaction of requirements may not be possible. Therefore, in computer animation we try to come up with techniques whereby we can automate parts of the process of creating animations that meet the given requirements.

Generating images that are required to look realistic is normally considered the precept of computer graphics, so computer animation has focused on the *low-level* realistic locomotion problem. For example, "determine the internal torques that expend the least energy necessary to move a limb from one configuration to another" is an example of a low-level control problem. While there are still many open problems in low-level control, researchers are increasingly starting to focus on other requirements such as "characters should behave realistically". By this we mean that we want the character to perform certain recognizable sequences of gross movement. This is commonly referred to as the *high-level* control problem. With new applications, such as video games and virtual reality, it seems that this trend will continue.

In character animation and in computer game development, exerting high-level control over a character's behavior is difficult. A key reason for this is that it can be hard to communicate our instructions. This is especially so if the character does not maintain an explicit model of its view of the world. Maintaining a suitable representation allows high-level intuitive commands and queries to be formulated. As we shall show, this can result in a superior method of control.

The simplest solution to the high-level control problem is to ignore it and rely, entirely, on the hard work and ingenuity of the animator to coax the computer into creating the correct behavior. This is the approach most widely used in commercial animation production. In contrast, the underlying theme of this document is to continue the trend in computer animation of building *computational models*. The idea is that the model will make the animator's life easier by providing the right level of abstraction for interacting with the computer characters. The computational aspect stems from the fact that, in general, using such models will involve shifting more of the burden of the work from the animator to the computer. Figure 1.1 gives a graphical depiction of this process.

1.1 Previous Models

In the past there has been much research in computer graphics toward building computational models to assist an animator. The first models used by animators were *geometric models*. Forward and inverse kinematics are now widely used tools in animation packages. The computer maintains a representation of

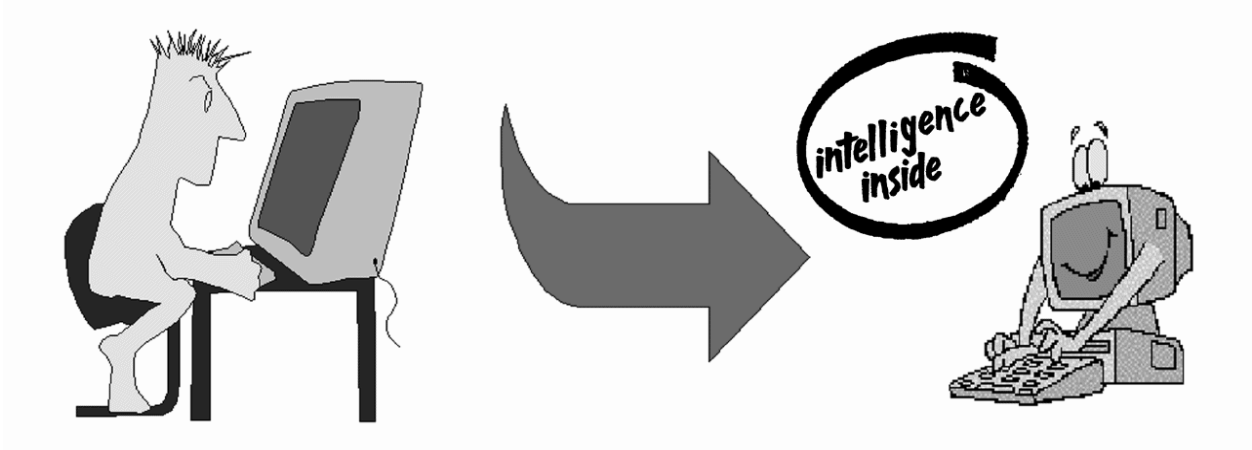


Figure 1.1: Shifting the burden of the work.

how parts of the model are linked together and these constraints are enforced as the animator pulls the object around. This frees the animator from, necessarily, having to move every part of an articulated figure individually.

Similarly, using the laws of physics can free the animator from implicitly trying to emulate them when they generate motion. *Physical models* are now being incorporated into animation packages. One reasonable way to do this is to build a computer model that explicitly represents intuitive physical concepts, such as mass, gravity, moments of inertia etc.

Physical models have allowed the automation of animating passive objects, such as falling chains, and colliding objects. For animate objects an active area of research is how to build *biomechanical models*. So far, it has been possible to use simplified biomechanical models to automate the process of locomotion learning in a variety of virtual creatures, such as fish, snakes, and some articulated figures.

Our work comes out of the attempt to further automate the process of generating animations by building *behavior models*. Within computer animation, the seminal work in this area was that of Reynolds [96]. His "boids" have found extensive application throughout the games and animation industry. Recently the work of Tu and Terzopoulos [114], and Blumberg and Galyean [23], has extended this approach to dealing with some complex behaviors for more sophisticated creatures. The idea is that the animator's role may become more akin to that of a wildlife photographer. This works well for background animations. For animations of specific high-level behaviors, things are more complicated.

We refer to behaviors that are common in many creatures and situations as *low-level* behaviors. Examples of low-level behaviors include obstacle avoidance and flocking behavior. Behaviors that are specific to a particular animal or situation we refer to as *high-level* behaviors. Examples include creature-specific mating behaviors and "intelligent" behavior such as planning. Many of the *high-level behaviors* exhibited by the creatures in previous systems suffer from the problem of being hard-wired into the code. This makes it difficult to reconfigure or extend behaviors.

Some of the work done by the logical programming community has some overlap with our work. In section 2.6, we shall discuss some of the achievements and limitations of that field. The main problem, however, is the lack of any satisfactory model of a character's cognitive process. Consequently it might be hard for them to extend their work to deal with important issues such as sensing, and multiple agents.

1.2 Cognitive models

Cognitive models are the next logical step in the hierarchy of models that have been used for computer animation. By introducing such models we make it easier to produce animations by raising the level of abstraction at which the user can direct animated characters. This level of functionality is obtained by

enabling the characters themselves to do more of the work.

It is important to point out that we do not doubt the ability of skillful programmers to put together a program that will generate specific high-level behavior. Our aim is to build models so that skillful programmers may work faster, and, less skilled programmers might be afforded success incommensurate with their ability. Thus, cognitive models should play an analogous role as might physics or geometric models. That is, they are meant to provide a more suitable level of abstraction for the task in hand – they are not, *per se*, designed to replace the animator.

Building cognitive models is very much a research area at the forefront of artificial intelligence research. It was thus to cognitive robotics that we turned for inspiration [68]. The original application area was robotics but we have adapted their theory of action to related cognitive modeling problems in computer animation.

One of the key ideas we have adopted is that knowledge representation can play a fundamental role in attempting to build computational models of cognition. We believe that the way a character represents its knowledge is important precisely because cognitive modeling is (currently) such a poorly defined task. If a grand unifying theory of cognition is one day invented then the solution can be hard-coded into some computer chips and our work will no longer be necessary. Until that day, general purpose cognitive models will be contentious or non-existent. It would therefore seem wise to be able to represent knowledge simply, explicitly and clearly. If this is not the case then it may be hard to understand, explain or modify the character's behavior. We therefore choose to use regular mathematical logic to state behaviors. Of course, in future it may turn out that it is useful, or even necessary, to resort to more *avant-garde* logics. We believe, however, that it makes sense to push the simplest approach as far as it can go.

Admittedly, real animals do not appear to use logical reasoning for many of their decision making processes. However, we are only interested in whether the resulting behavior appears realistic at some level of abstraction. For animation at least, faithfulness to the underlying representations and mechanisms we believe to exist in the real-world are not what is important. By way of analogy, physics-based animation is a good example of how the real-world need not impinge on our research too heavily. To the best of our current knowledge the universe consists of sub-atomic particles affected by four fundamental forces. For physics-based animation, however, it is often far more convenient to pretend that the world is made of solid objects with a variety of forces acting on them. For the most part this results in motion that appears highly realistic. There are numerous other examples (the interested reader is referred to [35]) but we do not wish to wallow any further in esoteric points of philosophy. We merely wish to quell, at an early stage, lines of inquiry that are fruitless to the better understanding of this document.

1.3 Aims

By choosing a representation with clear semantics we can clearly convey our ideas to machines, and people. Equally important, however, is the ease with which we are able to express our ideas. Unfortunately, *convenience* and *clarity* are often conflicting concerns. For example, a computer will have no problem understanding us if we write in its machine code, but this is hardly convenient. At the other extreme, natural language is obviously convenient, but it is full of ambiguities. The aim of our research is to explore how we can express ourselves as conveniently as possible, without introducing any unresolvable ambiguity in how the computer should interpret what we have written.

1.4 Challenges

The major challenge that faced us in achieving our aims was that we wanted our characters to display elaborate behavior whilst, possibly, situated in unpredictable and complex environments. This can make the problem of reasoning about the effect of actions much harder. This is a possible stumbling block in the understanding of our work, so we want to make our point as clear as possible.

A computer simulated world is driven by a mathematical model consisting of rules and equations. A forward simulation consists of applying these rules and equations to the current state to obtain the new state. If we re-run a simulation with exactly the same starting conditions we expect to obtain exactly the

same sequence of events as the last time we ran it. Therefore, it might not be clear to the reader in what sense the character's world is "unpredictable". To explain, let us imagine a falling stack of bricks, the top one of which is desired by some character. In the real world, it is almost impossible to predict, with any accuracy, where the bricks will come to rest. It makes more sense to have the character wait (preferably at a safe distance) and see where the desired brick ends up. In a simulated world, we can, in principle, run the forward simulation once to see where the bricks end up. Then we can re-run the simulation and tell the character where the brick will end up. We can even give the character's themselves the ability to run forward simulations. Such a character would essentially be clairvoyant, it could pre-compute the final brick positions and go and quietly wait for its desired brick. The point we wish to make is that this approach is complicated, inefficient and (for non-clairvoyant characters) will result in unnatural behavior. Thus, the representation used for simulating the falling bricks is not necessarily the appropriate one for our character to have. Indeed, it is highly unlikely that the representation we use for simulating the character's world will be appropriate for the character's internal representation of that world used for deciding how to behave. A key consequence of this approach, however, is that events the character did not expect will occur in the characters' world. Hence, when we refer to the character's world as complex and unpredictable, we mean that it is so from the character's point of view.

It is worth pointing out that there should be nothing shocking in having more than one representation for the same thing. The practice is commonplace, and is central to computer graphics. For example, consider the process of rendering a geometric model. For building the model we may choose to represent the model as a parametric surface. To take advantage of commonly available graphics hardware accelerators we may then move to a representation in terms of strips of triangles in three dimensions. At the final stage of rendering the objects will be represented as rows of pixels of differing intensities. The point is that at each stage a different representation of the same thing is appropriate.

Having multiple characters in a scene opens up possibilities for cooperative and competitive behaviors. However, even in purely kinematic worlds, this greatly increases the difficulty of predicting future events. That is, if one character is going to know what all the other characters are going to do in advance then it needs to have a complete and accurate representation of all the other characters' internal decision making processes. Even worse, it must be able to predict how they all react with each other, with the environment, and to itself. If the reader remains unconvinced then we need only consider the addition of user interaction to completely dispel all hopes of a character being able to predict all elements of its environment with complete accuracy. Perhaps even more damning is the observation that super-intelligent characters that can peer into each other's minds and predict every event that occurs in their world are not desirable. That is, we suppose that the majority of animations will want to reflect the fact that real world creatures are not able to exactly predict the future.

To summarize, the key point is that in one animation the representation of the world used by each of the animated characters may vary, and it will not necessarily coincide with the representation for other purposes. Thus, when we say the world is "unpredictable", we mean that it is hard to predict using the representation the character has of that world. It had, of course, better be entirely predictable from the simulation viewpoint!

Moreover, even if all characters represent the same type of things, what each of them actually "knows" about the world may be quite different. That is to say each character will be *autonomous*. By making them independent, self-contained entities, we replicate the situation in the real-world and thus ensure the level of realism normally required in animations. We also simplify the task of instructing them since we need only concern ourselves with them one at a time. As we pointed out above the correlation between the character's representation of its world and the representation for simulation should not solely be maintained by reasoning. The solution is that *sensing* must play a crucial role.

Figure 1.2 depicts a scene in which a character is facing away from a light bulb. The character does not know if the light bulb is on or not. It imagines many possible worlds, in some of which the light is on, in some of which it is off. Of course, the worlds are also distinguished by the other things the character believes about its world. Perhaps, in some of the worlds, the character imagines it has selected a winning lottery number, in some of them not. Regardless, from the point of graphical rendering the light bulb is indeed switched on. The correspondence between the character's view of the world and what is actually the case (which is presumably what we want to render) can be established by the character turning around and

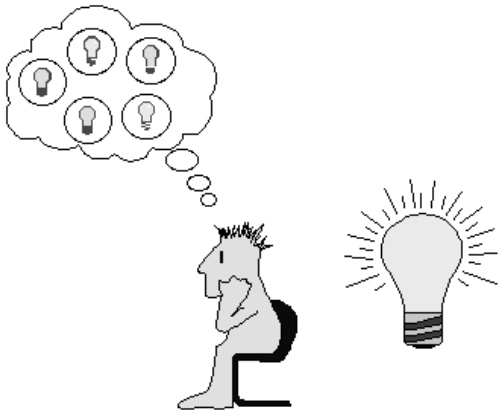


Figure 1.2: Many possible worlds.

looking. The act of sensing forces the character to discard, as impossible, the worlds in which it imagined the light was on. We shall return to this topic in more detail in section 3.8.

1.5 Methodology

We have achieved our aims for high-level control of animated characters by adopting an unambiguous semantics for a character's representation of its dynamic world. In particular, we propose an approach in which the user gives the character a behavior outline, or "sketch plan". The behavior outline specification language has syntax deliberately chosen to resemble that of a conventional imperative programming language. In terms of functionality, however, it is a strict superset. In particular, a behavior outline need not be deterministic. This added freedom allows many behaviors to be specified more naturally, more simply, more succinctly and at a much higher-level than would otherwise be possible. The character has complete autonomy to decide on how to fill in the necessary missing details. For example, with some basic background information we can ask a character to search for any path through a maze. That is, we do not initially have to give an explicit algorithm for how to find a particular path. Later we may want to speed up the character's decision making process by giving more detailed advice.

Although the underlying theory is completely hidden from the user, the success of our approach rests heavily on our use of a rigorous logical language, known as the *situation calculus*. Aside from being powerful and expressive, the situation calculus is well-known, simple and intuitive to understand. The basic idea is that a character views its world as a sequence of "snapshots" known as *situations*. An understanding of how the world can change from one situation to another can then be given to the character by describing what the effect of performing each given action would be. The character can use this knowledge to keep track of its world and to work out which actions to do next in order to attain its goals. The version of the situation calculus we use is inspired by new work in cognitive robotics [68]. By solving the well-known "frame problem", it allows us to avoid any combinatorial explosion in the number of action-effect rules. In addition, this new work incorporates knowledge-producing actions (like sensing), and allows regular programming constructs to be used to specify sketch plans. All this has enabled us to propel our creatures out of the realm of background animation and into the world of character animation. Finally, there is active research into extending the expressiveness of the situation calculus, and this makes it an exciting choice for the future.

We have developed a character design workbench (CDW) that is both convenient to use, and results in executable behavioral specifications with clear semantics. We can use the system to control multiple characters in realistic, hard to predict, physics-based, dynamic worlds. Interaction takes place at a level

that has many of the advantages of natural language but avoids the associated ambiguities. Meanwhile, the ability to omit details from our specifications makes it straightforward to build, reconfigure or extend the behavior control system of the characters. The use of logical reasoning to shift more of the burden for generating behavior from the animator, to the animated characters themselves, our system is ideal for rapid prototyping and producing one-off animations. Naturally, our system allows for fast replay of previously generated control decisions. However, when the speed of the initial decision making process is crucial, the user can easily assume more responsibility for efficiency. In particular, the behavioral controller can be gradually fine-tuned to remove the non-determinism by adding in more and more algorithmic details.

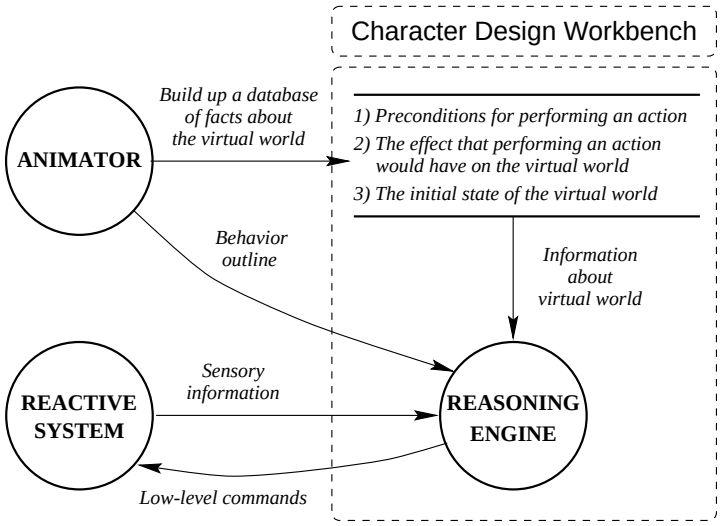


Figure 1.3: Interaction between CDW, the animator and the low-level reactive behavior system.

The potential of our approach is demonstrated by some intriguing animations of physics-based "merpeople" characters engaged in pursuit and evasion behaviors. Using our system meant that without having to state exactly which one, the merpeople were able to pick any goal position that met some given specification. The animations were the result of elegantly incorporating the existing lower-level reactive behavior system, described in [114], into CDW.

1.6 Overview

In chapter 2 we will provide a categorization of some previous work that is mainly concerned with modeling the virtual world for simulation purposes. This will be important for readers unfamiliar with computer animation. It will provide the necessary background information to understand the ideas that follow. We will also show how behavior animation came out as a natural consequence of continuing animation research. We will talk about some previous work and show how it fits into computer animation research as a whole.

Chapter 3 will discuss the theoretical foundations of our work. We shall discuss previous work from which we obtain the basic concepts. These concepts will be defined and explained with examples. Particular attention will be paid to our approach to sensing to deal with the correspondence between the agent's representation of its world and the representation the computer maintains for simulation purposes.

The predictability of kinematic worlds means that our character-oriented view of the world, and our simulation-oriented view of the world may coincide. In chapter 4 we will show how the situation calculus can be used to control a character's behavior in a very direct way. That is, it can be conveniently used right down to the locomotion level. We shall demonstrate how the nondeterminism in our specification language can be used to succinctly specify behaviors. An interpreter for our language can then automatically search for behaviors that conform to the specification. We conclude the chapter with a discussion of an exciting application of our work to cinematography.

Naively applying the situation calculus to physics-based applications leads to problems. In particular it is unlikely that we would want to produce a complete axiomatization of the virtual worlds' complicated causal laws as embodied in the state equations. Therefore we use the situation calculus to model the agent's high-level knowledge of some of the relevant casual relationships in its domain. For the sake of realism, efficiency, or both, we may choose to leave some of those relationships unspecified. For example, we probably don't want to axiomatize the laws of physics by which a ball in our domain moves. It thus becomes imperative that we incorporate sensing to obtain information about those aspects we choose to omit. Chapter 5 therefore seeks to exemplify the use of sensing to allow reasoning in unpredictable worlds. We give an example application of a physics-based underwater simulated world that is populated by, among other things, merpeople. The underlying physical model is that of [113]. The underlying model gives us the required level of unpredictability and allows us to produce visually appealing animations. Currently the merpeople can hide from predators (e.g. a large shark) behind obstacles that they "reasons" will obscure it from the predator's view.

Chapter 6 concludes the document, with a reverberant look at what has been accomplished and points out some promising directions for future work.

Chapter 2

Background

In this chapter we shall give a brief overview of computer animation. This will enable us to carefully position our work and give the necessary background information to understand the terminology we will be using. We shall embark on our explanation by discussing geometric models, then we shall move on to physical models, biomechanical models and, finally, behavioral models. The remainder of the document can then be seen as proceeding from this discussion by describing our contribution to enforcing high-level behavior constraints through the use of cognitive models.

2.1 Kinematics

In the most general case the location and the shape of an object can change with time. A general scheme for describing the shape and location of an object was given in [111]. In particular, let each point in some object Ω (without loss of generality $\Omega = [0,1]^n$) be named by its *material* (or *intrinsic*) coordinates $\mathbf{u}$. Then for some given $\mathbf{u} \in \Omega$ and time $t \in \mathfrak{T}$ the corresponding position in $\mathbb{R}^3$ can be given by specifying the function $\mathbf{q} : \Omega \times \mathfrak{T} \to \mathbb{R}^3$.

Intuitively, $\mathbf{q}(\mathbf{u}, t)$ describes the position of each point in the object as a function of time. Thus the object is free to change position and shape without constraint.

2.1.1 Geometric Constraints

A geometric constraint is a method of stating that there are forces present that cannot be specified directly but are known solely in terms of their effect on the motion of the system. Such constraints pose two problems:

1. The coordinates are no longer independent;
2. The forces required to realize a constraint are not furnished *a priori*; they are among the unknowns that must be calculated or eliminated.

If a (possibly time-dependent) constraint can be written in the form: $f(\mathbf{q}_1, \mathbf{q}_2, \dots) = 0$, where $\mathbf{q}_i = \mathbf{q}_i(\mathbf{u}_i, t)$, then the constraint is said to be *holonomic*, otherwise it is said to be *nonholonomic*. Other constraints will be considered as they arise but first consider one of the most common examples of a (holonomic) constraint, that of rigid body motion. It can be expressed by equations that state that the distance between any two points in the body remains constant over time: $\forall t_0, t_1 \in \mathfrak{T}; \mathbf{a}, \mathbf{b} \in \Omega\ (|\mathbf{q}(\mathbf{a}, t_0) - \mathbf{q}(\mathbf{b}, t_0)| - |\mathbf{q}(\mathbf{a}, t_1) - \mathbf{q}(\mathbf{b}, t_1)| = 0)$.

2.1.2 Rigid Body Motion

The problem of working out the effects of the rigid body constraints on the allowable motion is resolved (it turns out that the internal forces of constraint cancel each other out) by Chasles' theorem which states that

the most general displacement of a body satisfying the rigid body constraints is a translation plus a rotation[1]. This produces a simple and convenient way of representing rigid body motion in terms of homogeneous transformations (see appendix B for a definition of homogeneous coordinates and transformations).

A rigid body can be located in space (see [47] for a detailed discussion) by fixing, relative to the coordinate axes of some external coordinate system $\mathcal{C}_0$, a local coordinate system $\mathcal{C}_1$ inside the rigid body. Let $\mathbf{S}_0^1(t)$ be the homogeneous transformation matrix that maps $\mathcal{C}_0$ at time 0 into $\mathcal{C}_1$ at time t, so that $\mathbf{S}_0^1(0) = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix. A user can now specify the amount the object should be rotated or translated within some time Δt and the rigid body can be moved by automatically forming the corresponding homogeneous transformation $\mathbf{C}$ and premultiplying it by $\mathbf{S}_0^1(t)$ to give the new homogeneous transformation $\mathbf{S}_0^1(t + \Delta t) = \mathbf{C}\ \mathbf{S}_0^1(t)$. So, assuming $\mathbf{q}(\mathbf{u}, 0)$ is given, $\mathbf{q}(\mathbf{u}, t) = \mathbf{S}_0^1(t)\mathbf{q}(\mathbf{u}, 0)$.

2.1.3 Separating Out Rigid Body Motion

In [110] the above ideas on rigid body location were incorporated into the more general formulation that allows for deformation of shape. The basic idea was to produce a hybrid model with explicit deformable and rigid components. Thus the body has its own coordinate frame $\mathcal{C}_1$ whose origin coincides with the body's center of mass $\mathbf{c}(t)$. The movement of $\mathcal{C}_1$ (represented by the transformation matrix $\mathbf{S}_0^1$) accounts for the rigid body motion, while the component due to shape deformation is represented with respect to $\mathcal{C}_1$ by a reference component $\mathbf{r}$ and a displacement component $\mathbf{e}$: $\mathbf{q}(\mathbf{u}, t) = \mathbf{S}_0^1(\mathbf{r}(\mathbf{u}, t) + \mathbf{e}(\mathbf{u}, t))$ (see Figure 2.1). Note that many instances of $\mathbf{S}_0^1$ and $\mathbf{e}$ exist that account for the same shape.

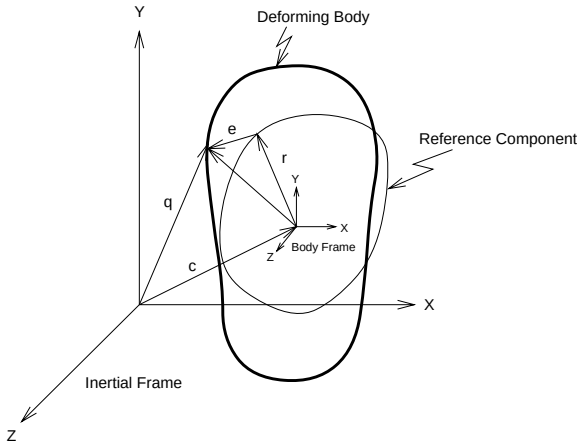


Figure 2.1: Kinematics

2.1.4 Articulated Figures

By generalizing the ideas in section 2.1.2 it is possible to describe *articulated figures*. Many things, such as the human skeleton, can be modeled as an articulated figure. An articulated figure consists of a number of objects, known as *links*, connected by constraints, known as *joints*. In general it is possible to have nonholonomic joint constraints but here it will be sufficient to consider holonomic joint constraints. A holonomic joint corresponds to the removal of one or more "degrees of freedom" from the object's allowable range of motion. Such a joint can be decomposed, without loss of generality, into a set of prismatic joints and a disjoint set of revolute joints. As the name suggests, prismatic joints move by translating in a plane, while revolute joints move by rotating about an axis.

Consider the case of an articulated figure that consists of a chain of links each connected by a joint. For an n-axis articulated figure there are $n + 1$ links (link 0 is the *base* and link n is the *end-effector*) connected

[1]It is possible to pick the body-set of coordinates to give a rotation about the direction of translation. This *screw* motion is much used in robotics.

by n joints. Each joint has a set of *joint parameters* associated with it and each link has a corresponding set of *link parameters* (see figure 2.2).

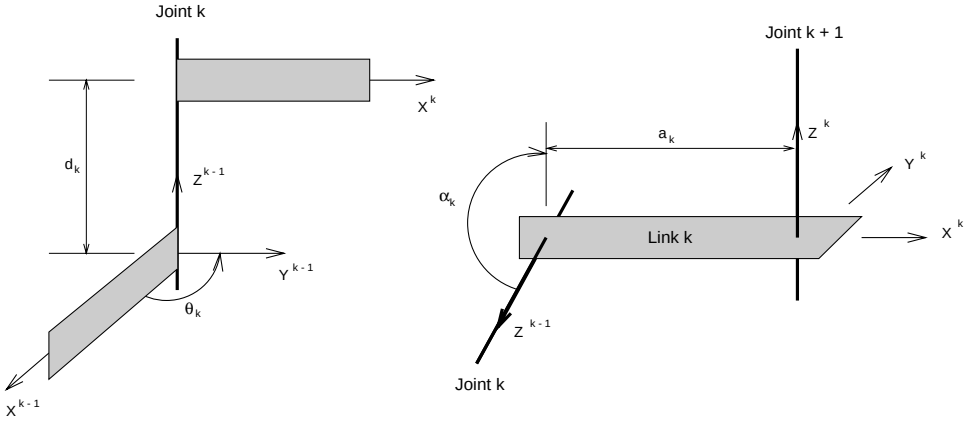


Figure 2.2: Joint and Link Parameters

Let joint k connect link $k-1$ to link k, and define the kth *joint variable* as a function of time t: $l_k(t) = \xi_k \theta_k(t) + (1-\xi_k) d_k(t)$, where $\theta_k(t)$ is the angle of joint k at time t, $d_k(t)$ is the linear displacement of joint k at time t, and ξ_k is a function such that:

$$\xi_k = \begin{cases} 1 & \text{if joint } k \text{ revolute,} \\ 0 & \text{if joint } k \text{ prismatic.} \end{cases}$$

Using the Denavit-Hartenberg notation [34], say, assign, for $0 \leqslant k \leqslant n$, a link coordinate frame $\mathcal{L}_k$ to the distal end of each link ($\mathcal{L}_n$ ends up at the tip).

Let $\mathbf{T}^k_{k-1}$ be a function of $l_k(t)$; that is, if the user specifies the amount a link should be rotated or translated within some time Δt then $\mathbf{T}^k_{k-1}(l_k(t+\Delta t))$ represents the corresponding homogeneous transformation that gives the position of points in the kth link in terms of $\mathcal{L}_{k-1}$.

Apropos section 2.1.2, let $\mathcal{L}_0 = \mathcal{C}_1$ and let $\mathbf{S}^1_0(t)$ represent the rigid body motion of the whole articulated figure. Then given a complete vector of joint variables $\mathbf{l}(t) = (l_0(t), \ldots, l_n(t))$ it is possible to calculate the position of the end-effector, at time t in terms of $\mathcal{C}_0$: $\mathbf{S}^n_0(\mathbf{l}) = \mathbf{S}^1_0\ \mathbf{T}^1_0(l_1) \cdots \mathbf{T}^n_{n-1}(l_n)$. Problems arise with articulated figures containing closed loops (see [39]), but with suitable relabeling all of the above extends to articulated figures with more general topologies.

Thus, since link k's coordinate frame is defined in terms of link $k-1$'s coordinate frame, rotating or translating link $k-1$, causes all the $j \geqslant k$ links to move by the same amount. This makes direct manipulation of articulated figures easier.

Forward Kinematics

Let the *joint space* $\mathfrak{L} \subset \mathbb{R}^n$ of an articulated figure be the set of all possible joint vectors, and let the *configuration space* $\mathfrak{Z} \subset \mathbb{R}^6$ be the set of all possible configurations (position plus orientation) of the end-effector. Then the *forward kinematics* problem is to determine the function $\mathbf{w} : \mathfrak{L} \to \mathfrak{Z}$ that maps joint vectors to configuration vectors.

The means to calculate a solution to the forward kinematics problem have already been outlined in section 2.1.4. Such a solution can be conveniently expressed in the form:

$$\mathbf{S}^n_0(\mathbf{l}) = \begin{pmatrix} \mathbf{R} & \mathbf{p} \\ \mathbf{0} & 1 \end{pmatrix}$$

where $\mathbf{R}$ and $\mathbf{p}$ represent, respectively, the orientation and position of l_n in $\mathcal{C}_0$ at time t.

Inverse Kinematics

Many problems in animation (and robotics) are naturally phrased as constraining the end-effector to be in some configuration. The *inverse kinematics* problem is to determine the function $\mathbf{w}^{-1} : \mathfrak{Z} \rightarrow \mathfrak{L}$ that maps configuration vectors to joint vectors.

It is possible to cast this problem as an optimization problem and, in the graphics literature [7, 8], describe a numerical technique in which the user is given interactive control over how much time the algorithm spends on trying to compute a solution. For a specific geometric model it may also be possible to obtain a closed-form solution. Unfortunately no general technique exists for doing this and, even when such a solution does exist, it is usually difficult to derive.

An alternative approach is to examine the differential relationship (with respect to time) between $\mathbf{l}(t) \in \mathfrak{L}$ and $\mathbf{z}(t) \in \mathfrak{Z}$: $\dot{\mathbf{z}} = \mathbf{J}(\mathbf{l})\dot{\mathbf{l}}$, where $\mathbf{J}$ $(\mathbf{l})$ is known as the *Jacobian matrix* of the end-effector:

$$J_{kj} = \frac{\partial w_k(\mathbf{l})}{\partial l_j} \quad 1 \leqslant k \leqslant 6,\ 1 \leqslant j \leqslant n.$$

This matrix effects a linear transformation that maps instantaneous joint space velocities into instantaneous configuration space velocities. This differential relationship can be used as the basis for numerical methods that can solve for a joint space *trajectory* in terms of a given configuration space trajectory. These techniques can also be extended to handle over-constrained and under-constrained problems (see [99]). How a trajectory, in any space, might be computed to start with is discussed in section 2.2 onwards.

2.2 Kinematic Control

2.2.1 Key-framing

Suppose an animation consists of k frames, each of which depicts the position and shape of some object at times $t_i \in \{t_0, \ldots, t_{k-1}\} \subset \mathfrak{T}$. Let $\mathbf{q} : \Omega \times \mathfrak{T} \rightarrow \mathbb{R}^3$ be a user defined partial function that specifies the position of any point on the object at the given times. Then for each t_i the set of values $\{\mathbf{q}(\mathbf{u}, t_i) | \mathbf{u} \in \Omega\}$ is known as a *key-frame*.

The function $\mathbf{q}$ need not be specified directly but can be calculated (using the techniques outlined previously) from, say, a trajectory through joint space. At the practical level users usually define key-frames using interactive shape manipulation (assuming there is no rigid body motion constraint), forward and, if available, inverse kinematics procedures. Once the scene is in the desired configuration it is recorded and the process repeated until as many scenes as required are defined.

A widely used interactive direct manipulation technique is to have an actor kitted out with some motion detectors and have the movements of the actor mapped, in some useful way, to a corresponding object in the scene. While this ensures very realistic looking motion, it says nothing about the underlying mechanisms, it is hard to apply to figures with a different topology to the one used for the motion detection, and is hard to modify in a realistic way. There has even been some work [27] on trying to adapt dance notations (for example Labanotation) to develop a language for describing these motions. Unfortunately while these "languages of movement" may have some merit as regards description they have little use as an animation synthesis tool. It is simply too complicated and counterintuitive for a non-expert to script or alter animations using the low-level constructs available in these languages.

The transition from a discrete set of samples through the space in question to a smooth path is achieved by interpolation. In the above example, for some given $\mathbf{u}$, $\mathbf{q}$ is made a total function by using the set of values $\{\mathbf{q}(\mathbf{u}, t) | t \in t_0, \ldots, t_{k-1}\}$ as control points for a space-time spline [106] (most modern day animation systems use interpolating Catmull-Rom splines). In conventional animation this process of interpolation is commonly known as *in-betweening* and was quickly adopted by computer animation systems by making the computer responsible for producing the in-between frames [28]. Naively applying the same strategy to object orientations can lead to non-smooth rotations and so [101, 17] gave methods for using interpolation of quaternions that produces the desired smooth rotations.

Key-framing is a very flexible approach but a major drawback is that the length and realism of an animation is usually proportional to the amount of effort expended by the animator. However it remains the most commonly used approach in commercial animation design today ([62] gives some helpful guidelines to would-be computer animators) and can easily be used to produce popular effects such as shape "morphing."

2.2.2 Procedural Control

There are many other ways to non-interactively define a trajectory through some space using principally kinematic methods. In particular [21, 95] showed how a general purpose programming language (with some extensions for animation purposes) could be used to define arbitrarily complicated trajectories. Indeed the literature abounds with such definitions, usually they have been hand-crafted to solve some particular problem and sometimes they contain elements that are useful in a wider context. To name but a few: [7] used a network of special purpose processors to produce animations of a human-like figure; [127] created a hierarchical animation system that used finite state machines to generate walking and jumping motions; [45] used a mixture of inverse kinematics, simple rules and some simple dynamics to create some very realistic animation of human walking; [65] gave an account of the robot motion planning problem and expounded a solution using commonly available graphics techniques; [97] used a combination of rules and inverse kinematics to calculate realistic grasps (this has also been the subject of much research in robotics); [90] described how multiple kinematic constraints can be applied to a three-dimensional human body model, so that it can be interactively manipulated with much more ease than would otherwise be possible; [61] described a manipulation motion planning system inspired by robotics research; and [8] discussed human modeling using mainly kinematic methods.

2.3 Noninterpenetration

A noninterpenetration constraint states that the intersection of two bodies can never include more than their boundaries. That is, for some object A define F_A such that at any time $t \in \mathfrak{T}$ and for any given point $\mathbf{q} \in \mathbb{R}^3$:

$$F_A(\mathbf{q},t)\begin{cases} > 0 & \text{if } \mathbf{q} \text{ is outside } A, \\ = 0 & \text{if } \mathbf{q} \text{ is on the boundary of } A, \\ < 0 & \text{if } \mathbf{q} \text{ is inside } A. \end{cases}$$

For any two bodies A and B whose positions are, respectively, specified by functions $\mathbf{q}_A$ and $\mathbf{q}_B$ a *noninterpenetration* is maintained by satisfying the nonholonomic constraint:

$$\forall t \in \mathfrak{T}; \mathbf{u}_A \in \Omega_A; \mathbf{u}_B \in \Omega_B \ (F_A(\mathbf{q}_B(\mathbf{u}_B,t),t) \geqslant 0 \ \wedge \ F_B(\mathbf{q}_A(\mathbf{u}_A,t),t) \geqslant 0)).$$

The first part of the problem, known as the *collision detection problem*, is to determine the time at which any two bodies first collide. That is, we seek a t for which equality in the above expression is reached. The problem of determining the subsequent motion is known as the *collision resolution* and *resting contact* problem.

2.3.1 Collision Detection

Collision detection is a geometry problem and has been extensively studied in robotics and computer-aided design as part of the *collision avoidance* problem. The different demands of a collision detection algorithm for computer animation led [81] to study the problem in the context of computer animation. In particular an algorithm is given that uses point sampling at a given time to find a set of interpenetrating points for time dependent parametric surfaces and convex polyhedra[2]. A similar approach is used in [54] but the algorithm also searches for the time that the first collision occurred. In [11, 12], for bodies composed of both polyhedra and convex closed curved surfaces, temporal coherence is exploited to achieve faster average running times.

[2] Concave polyhedra may be decomposed into a set of convex polyhedra.

With the above approaches, if an object goes right through another object, between the times for which collisions are checked, then the algorithm will fail. As argued in [81], this can be made unlikely by checking regularly but this may be inefficient, especially if there is only a small number of collisions. So for time dependent parametric surfaces, [120] uses the idea of a (user-definable) near-miss and analysis of derivatives for each surface type to avoid missing collisions. An approach based on interval analysis is presented in [103, 104]. The algorithm is reported to be robust, efficient, accurate to within a user specified tolerance and applicable to a wide range of time dependent parametric and implicit surfaces. One of the most comprehensive treatments of collision detection can be found in [71].

2.3.2 Collision Resolution and Resting Contact

When two rigid bodies first come into contact, they exert a large force on each other for an infinitesimally small amount of time. This collision force, known as an *impulse*, results in a discontinuous change of velocity. The bodies may then remain in resting contact where *resting contact forces* prevent subsequent interpenetration. In [81] the nondeterministic problem of multiple simultaneous collisions is dealt with by a propagation method while [11] uses a simultaneous approach that is more efficient.

In [81] an analytic method that uses the principle of conservation of momentum to calculate the effects of impulsive forces on rigid bodies (including articulated figures) is given. In addition, for rigid and deformable bodies, a nonanalytic *penalty method*, equivalent to inserting a spring and damper between any two contact points, is described. Resting contact is modeled as a sequence of collisions. For deformable bodies [92, 110] introduce arbitrary penalty forces between colliding bodies to separate them.

Preventing interpenetration with penalty methods is slow and not necessarily physically correct so a series of papers [11, 12, 13, 16, 14] sought to further investigate the use of analytic methods. The first step was to reformulate the noninterpenetration constraint for curved surfaces in a form that was specific enough to be differentiable. Then (in [11, 12, 13]) the problem of computing contact forces between a pair of bodies that contact at a point without friction was considered. Next systems of bodies that behave according to the classical Coulomb model of friction was discussed. This leads to systems in which there are no solutions to the classical constraint force equations, as well as systems that admit multiple solutions for the constraint force equations and whose subsequent behavior is thus indeterminate. Both computational and practical complexity results for simulating such systems were given. In [14] an alternative iterative approach was presented that is claimed to be faster, more reliable, applicable to the case of dynamic and static friction, and simple enough to be implemented by a nonexpert in numerical algorithms.

In [16] the analytic approach to noninterpenetration is applied to flexible bodies that are restricted to only undergo shape deformations that can be expressed as a global deformation. There is a problem with exactly determining the contact surface, which [42] solves by using a purely kinematic step in an implicit formulation of the notion of a deformable body.

2.4 Dynamics

Some [90, 61] have pointed out that for low-speed motion the probability of producing physically implausible looking motion is low and they even suggest using simple qualitative physics notions to improve the look of faster motion. However all kinematic approaches to producing motion underconstrain the allowable motion so that objects may be allowed to move in a completely unrealistic way.

The laws of classical physics constitute a precise statement of people's preconceptions about how everyday objects are expected to move. Some physics that is relevant to computer animation has already been described in section 2.3.2. What follows describes the physics that is applicable to generating motion in general and discusses some of the implementation issues that arise.

2.4.1 Physics for Deformable Bodies

The physically realistic simulation of deformable bodies for animation was first addressed in [111]. The equations of motion can be written in Lagrange's form as

$$\frac{\partial}{\partial t}(\mu\frac{\partial \mathbf{q}}{\partial t}) + \gamma\frac{\partial \mathbf{q}}{\partial t} + \frac{\delta\varepsilon(\mathbf{q})}{\delta \mathbf{q}} = \mathbf{f}(\mathbf{q}, t),$$

where $\mathbf{q}(\mathbf{u}, t)$ is the position of point $\mathbf{u}$ at time t, $\mu(\mathbf{u})$ is the mass density, $\gamma(\mathbf{u})$ is the damping factor, $\mathbf{f}(\mathbf{q}, t)$ is the net externally applied force and $\varepsilon(\mathbf{q})$ is a functional that measures the net instantaneous potential energy of the body. $\delta\varepsilon(\mathbf{q})/\delta\mathbf{q}$ is the variational derivative that measures the rate of change of the potential energy with respect to the deformation.

In [110] the corresponding equations for the hybrid model of section 2.1.3 were given (the different representations give different practical benefits at the extremes of deformable behavior). The definition of $\varepsilon(\mathbf{q})$, based on the theory of elasticity, allows for objects to display elastic and inelastic motion. Through a discretization, based on the finite elements method, and the application of numerical integration techniques to the solution of the equations of motion, a computer implementation was possible. The discretization is equivalent to a model consisting of a number of point masses connected by springs. Such a model has been successful in modeling certain classes of animals such as snakes [80] and fish [114]. To produce more efficient implementations the allowable deformations can be restricted as in [88, 124, 38, 128]. It is also possible to have articulated bodies with deformable links [79].

2.4.2 Physics for Articulated Rigid Bodies

There are two commonly used approaches to deriving the equations of motion for a rigid articulated body: the Lagrange-Euler formulation and the Newton-Euler formulation. Both result in the same motion (indeed one may be derived from the other [47]) but they have different computational properties.

Lagrange's Equation

The Lagrange-Euler formulation is based on the concepts of generalized coordinates, energy and generalized force (see [47] for an explanation). The approach makes the forces that maintain geometric constraints implicit, thus resulting in a reduced set of coordinates. This has the advantage that complex dynamic systems can sometimes be modeled in a simple, elegant fashion with a minimal set of coordinates. In addition, the terms in the final closed form equations often have simple physical interpretations.

For an n-link articulated figure an appropriate set of generalized coordinates is the vector of joint variables. The generalized coordinates are chosen so that they are, at least for the case of holonomic constraints, independent. In particular, for articulated figures with no loops there are well known $O(n)$ methods for computing the accelerations of the joints [39]. For figures containing loops, and other nonholonomic constraints, it is not necessarily possible to derive a suitable set of generalized coordinates. Even if a set of coordinates can be found, computing the accelerations has worst case complexity of $O(n^3)$.

Newton-Euler Formulation

Unlike the Lagrange formulation, the Newton-Euler approach does not use the notion of generalized coordinates. This results in a full set of coordinates that are no longer necessarily independent and for which the (explicit) forces of constraint must be determined. For a rigid body whose center of mass is given by $\boldsymbol{c}(t)$, the basic Newton-Euler equations of motion are:

$$\sum \mathbf{f} = \mathbf{m}\ddot{\mathbf{c}} \quad \text{and} \quad \sum \boldsymbol{\tau} = \mathbf{I}\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times \mathbf{I}\boldsymbol{\omega},$$

where $\mathbf{f}$ is the external force, $\mathbf{m}$ is the mass, $\ddot{\mathbf{c}}$ is the acceleration, $\boldsymbol{\tau}$ is the external torque, $\mathbf{I}$ is the inertia tensor, $\boldsymbol{\omega}$ is the orientation and $\dot{\boldsymbol{\omega}}$ is the angular velocity.

The main contribution of study in the dynamics of articulated rigid bodies has been to come up with fast $O(n)$ recursive solutions based on the Newton-Euler formulation [39]. The recursive formulation exploits the chainlike structure of articulated figures, with the motion of each link represented with respect to its neighboring link. Given a joint-space trajectory, the velocities and accelerations of each link are computed recursively, starting at the base and propagating forward to the tip. This produces the *forward equations*, which are then used to compute the forces and torques acting on each link, starting at the tip and working backwards to the base (the *backward equations*).

Recently, [15] described a direct (i.e. nonrecursive, noniterative) technique based on Lagrange multipliers. The Lagrange multipliers are the unknown constraint forces. Since internal forces do no net work we want to find the solution such that the internal forces of constraint disappear. This amounts to solving a large sparse matrix, which by exploiting its structure, can always be done in linear time. The approach handles arbitrary constraints, is conducive to good software design, is simple to understand and implement. When there are no cycles the method has worst case complexity of $O(n)$.

2.4.3 Forward Dynamics

The forward dynamics problem is that of computing positions and velocities from given forces and torques. The unknowns in the equations of motion of the previous sections are the forces, torques and accelerations. So the first part of the problem is to solve, in terms of the forces and torques, the equations of motion for the accelerations.

In general, this gives a set of coupled second order nonlinear differential equations. Any set of n-th order differential equations can be reduced to an equivalent first-order one of the form: $\dot{\mathbf{x}} = \mathbf{k}(\mathbf{x}(t), \mathbf{v}(t))$, where $\mathbf{v}(t)$ supplies values for all the unknown forces and torques. With suitable initial (or boundary) values $\mathbf{x}(t_0) = \mathbf{x}_0$ this system of equations can be numerically integrated over some interval $[t_0, t_1]$ to give positions and velocities $\mathbf{x}(t)$. A survey of numerical integration techniques for computer animation is given in [48].

Much early work in animation (and robotics) was concerned with producing programs that embodied the equations of motion for rigid body articulated figures [122, 5]. Software now exists that can take a physical description of some articulated figure and produce the corresponding equations of motion (for example the "dynamics compiler" in [118]). Such software is now available in robust and efficient commercial *simulators* such as [109]. Thus, for inanimate objects it is possible to create realistic motion in a completely automatic way by using *simulation*. All the animator need do is define the physical object descriptions, set the initial conditions and watch the animation unfold.

The situation is slightly complicated by the possibility that objects may collide. If a nonanalytic collision resolution algorithm is used, the springs and dampers can either be incorporated when a collision is detected (see section 2.3.1) or, if all the points of possible contact can be surmised in advance, they can be included *a priori* in the equations of motion. If an analytic method is used in which a discontinuous change of velocity occurs, then when the simulator detects a collision it suspends the simulation, resolves the collision (see section 2.3.2) and then restarts the simulation with the new initial conditions.

2.4.4 Inverse Dynamics

It is also possible to solve the equations of motion for the forces and torques in terms of the accelerations. Then given some desired motion it is possible (by twice differentiating) to calculate these accelerations to, in turn, calculate the forces and torques that would be required to produce such a motion.

In robotics the inverse dynamics formulation can be useful for calculating the torques necessary to move a robot along some specific path. In animation the goal is determining the motion, so if this is already known there is not much need to calculate the forces and torques that will produce it. Exceptions to this are when the forces and torques are used to evaluate the feasibility of some defined motion, say for ergonomics [8]; or when a mixture of kinematic and dynamics is used in one animation (see section 2.4.5).

2.4.5 Additional Geometric Constraints

Several authors have considered the problem of adding other types of constraints to a physical simulation: [18] presented a menagerie of possible constraints for rigid bodies; [92] considered the application of constraints to deformable models.

The point of the constraint based approach is to ease the problem of controlling a dynamic simulation, that is part of the simulation can be controlled kinematically (for example the endpoint of an articulated figure can be constrained to follow some path) and the rest of the simulation will move in accordance with the laws of physics. By using inverse dynamics to calculate the forces needed to enforce the constraint the effects of these forces can be incorporated into the physics based part of the animation. This approach to animation is suggested by [58] in the context of articulated figures.

2.5 Realistic Control

Most interesting animations will be of *actuated* (or animate) objects, that is objects that can exert some control over their motions. Up until now the only approach that has been mentioned for producing animations of actuated objects has been simply to state their positions at given times. The point is that in the real world an animal cannot produce arbitrary forces and torques (as in the inverse dynamics approach) to enable it to follow any given path in space as it is constrained to move using the finite resources available in its muscles. Therefore realistic motion of actuated figures can only be achieved by either placing complete reliance on the skill of the animator to correctly interpret this limitation or by adopting this requirement and solving an associated control problem. This section will explore the second alternative for which some notions and nomenclature from control theory will be required, the reader is referred to appendix C for a brief overview.

2.5.1 State Space

In computer animation the state vector (see appendix C) is the vector that must contain all the quantities that are required to completely specify the position and velocities of all the points of all the objects in the scene. The system state equations are precisely the ones discussed in section 2.4.3 that are required to compute the forward dynamics solution.

2.5.2 Output Vector

The output vector (see appendix C) supplies the controller with all the information with which it must decide which action to perform. The controllers used in [123, 32, 72, 119] are all functions of time (open-loop controllers), while in [25, 118] functions of state (closed-loop controllers) are used. In [118] making the controller a function of the state is problematic as the state space grows exponentially as the objects become more complex. So in [117, 87, 102] closed-loop control functions of simple input sensors are used. Closed-loop controllers are more flexible in that their output is not fixed by the current time (as in an open-loop controller) but is a function of the current situation. An open-loop controller must be generated for each new scenario. Thus, where the cost of synthesizing a controller is prohibitive a closed-loop controller represents a much more sensible approach.

2.5.3 Input Vector

The actions the controller chooses to perform are represented by input vectors (see appendix C). As stated in section 2.4.3 in order to compute the forward dynamics solution (assuming the initial conditions are fixed) values must be supplied for all the unknown forces and torques. Therefore the input vector must supply these values.

Supplying actual torques and forces is counterintuitive so usually a simple muscle model is used. In general no attempt has been made to realistically model the intricate complexity of muscles and tendons (see [29, 86] for more biomechanically based muscle models). Instead the effects are often approximated with

a *spring and damper* which acts as a proportional-derivative controller that will tend (over time) to bring a joint, say, to some desired length (angle). As pointed out in [93], the spring and damper model emulates the ability of real animals to make their motion a lot more efficient (especially at high speeds) by using the springiness of their muscles and tendons to store energy from one movement to the next. Also as the spring extends further from its rest length the restoring force becomes larger, thus automatically enforcing joint limits. A nonlinear spring is even more effective for enforcing joint limits but can give rise to stiff differential equations.

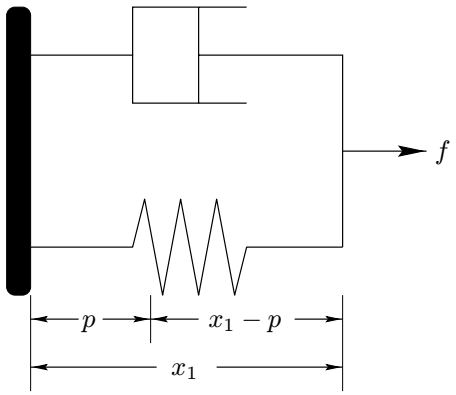


Figure 2.3: "Muscle" represented as a spring and damper

For a deformable model made up of point masses and springs, muscles can be modeled by designating some of the springs as actuated. This means that the input vector, as determined by some control function, is a reference setting for the equilibrium point of each actuated spring. Or, put more simply, the rest lengths of the springs are allowed to be changed by the control function. For an articulated figure an analogous (and common) technique is to insert a (possibly rotary) spring and damper at each joint. If the joint is designated as actuated then the rest length (angle) of the spring is again made the input value from some control function. By way of a concrete example, consider the case of a single "muscle and tendon" represented by a linear spring and damper (see figure 2.3). Let the input v (as determined by some control function p) represent the spring's rest length. Then, assuming the output vector is just the system state, the force exerted by the spring is $f = k_s(p(\mathbf{x}) - x_1) + k_d x_2$, where k_s is the spring stiffness, k_d is the damping factor, and $\mathbf{x} = (x_1, x_2)$ is the state vector (the spring length x_1 and velocity $x_2 = \dot{x}_1$). Incorporating this equation in the system state equations means that control can be exerted over the system by varying the rest length p.

2.5.4 Control Function

The control function (see appendix C) calculates values for the input vector from given output vectors.

Hand-crafted Controllers

One solution to producing a suitable control function to solve some control problem is simply to leave it up to some human to come up with. A typical example of a difficult control specification that can be *hand-crafted* in this way is one of requiring a controller to make an object move like some particular animal.

For deformable models [80, 114] presented hand-crafted controllers (for snakes and fish, respectively) that consist of parameterized sinusoidal contractions of spring rest lengths. There has been some success in hand-crafting controllers for articulated figures: [26] generated parameterized human walks that used a mixture of kinematics[3], dynamics and rules about gaits; [77] produced a dynamic simulation of a walking statically stable cockroach controlled by sets of coupled oscillators; [24] achieved similar results for a real-world six-legged creature; [93] showed how useful parameterized controllers of hoppers, kangaroos, bipeds,

[3]The kinematics is only used after the fact to improve the look of the motion.

and quadrupeds can be achieved by decomposing the problem into a set of simpler control problems; [107] created a dynamic simulation of a biped walking by defining a finite-state machine that adds and removes constraint equations. A good survey of these kinds of approaches can be found in [10].

Many of these approaches produce very useful parameterized controllers. The big drawback with hand-crafted solutions, other than that they can be extremely difficult to derive (especially for a complex articulated figure), is that they are not necessarily applicable to other systems. However, for obvious reasons, human beings are often the subject of computer animations and so hand-crafting a collection of useful controllers for a human like object is bound to be useful. It may even be possible to justify such an approach for other common animals.

Control Through Optimization

The optimal control approach uses optimization to try and produce an *optimal controller*. In producing an animation the evaluation criterion is largely an aesthetic one, which in most cases only induces a partial order on the state space trajectories. Therefore the choice of a performance index is largely subjective. For some problems a natural performance index exists, for instance if an animation is required of a creature doing a high jump then it is natural to choose a performance index that will prefer higher jumps over lower ones. In many situations it seems natural to assume that a solution that uses less energy and less time should be preferred. In practice a user-definable weighted combination of various criteria can be used to form a suitable performance index. By changing the weights the user can try and exert some rather indirect control over the look of the motion.

By using simple performance indices that reward low energy expenditure and distance traveled in a fixed time interval [117, 87, 102, 112, 119], motions have been produced that bear a distinct qualitative resemblance to the way that animals with comparable morphologies perform analogous (usually periodic) locomotion tasks. However it is not clear that suitable performance indices could be formulated *a priori* that could be predicted to produce something like, say, a "happy walk". Also, the approach can not deal with really dynamic factors.

Objective Based Control

It has already been seen in section 2.2 that a common way to specify the desired motion in a computer animation is by using key-frames. It was only natural therefore that some of the earliest approaches to realistic control in animation should look at ways of producing physically plausible motions that "interpolated" given key-frames [123]. Given that the animation starts in one key-frame, the other key-frames define a goal that must be achieved. In addition it is also necessary to concurrently satisfy the goal that the motion be in accordance with the laws of physics and that it be produced using the finite resources available in the creatures "muscles."

One approach that has been taken to solving these problems [123, 32, 72] has been to define a performance index and treat the problem as a constrained optimization problem. The idea is that an open-loop controller is synthesized by searching for values of the state space trajectory and the forces and torques that satisfy the goal and are minimal with respect to the performance index. This is achieved by numerical methods that iteratively refine a user supplied initial guess. The problem of dealing with overconstrained motion is obviated by giving the constrained optimization algorithm responsibility for arbitrating between the constraints (or goals). Unfortunately this means that for overconstrained problems compromise solutions can be produced that are not necessarily what the user wants. Even if the motion is not technically overconstrained the numerical algorithm may still not be able to find a solution that completely satisfies the constraints.

Often what the user wants is that no compromise should be made with the laws of physics. If this is so then a method must be chosen in which the laws of physics are incorporated in some inviolable way. A common way to do this [118, 117, 87, 102, 119] is to simply reformulate the problem as an *unconstrained* optimization problem (as in section 2.5.4). This means that the constraints are no longer hard and fast but rather the performance index is modified to reward motions the closer they come to satisfying the constraints.

2.5.5 Synthesizing a Control Function

There are some additional issues that arise in the process of synthesizing a control function:

Collisions and friction. Collisions and friction can have a detrimental effect on the numerical optimization algorithms that require access to derivatives [123, 32, 72]. This means that it is often necessary to formulate the problem so that arbitrary control, or no feedback control is exerted during periods of time in which collisions occur. In contrast the stochastic optimization techniques used in [117, 87, 102, 119] often produce trajectories that can be seen to take advantage of collisions and friction.

Generality. One of the first attempts to synthesize a controller for animation purposes was given in [25]. The paper employed techniques from optimal control theory for linear systems to create optimal controllers. Where an animation of a linear system is required this approach is ideal. However, control problems are made much harder by nonlinearities and, unfortunately, most interesting animations are highly nonlinear.

Local minima. In practice all numerical methods suffer from the problem of becoming trapped in local minima and thus it can not be guaranteed that the best solution has been attained. "Global" (stochastic) optimization algorithms can help in this regard [117, 87, 102]. Convergence to the global solution is guaranteed but the price paid is slow convergence, they are generally computationally intensive. Indeed [87, 102] avail themselves of a massively parallel computer to achieve faster results and [87] circumvent an aspect of the problem by controlling the limb placement kinematically. In [32] it is recognized that the application for the work is animation (as opposed to autonomous agents in robotics, say) and thus user interaction is allowed, even at the level of helping the optimization process out of local minima.

Post-processing. In [117] a stochastic generate-and-test procedure is used to learn the weights for a network of connections between binary sensors and actuators. These weights are then refined by a modify-and-test procedure that searches for better solutions in the immediate vicinity. For deformable models [52] uses a post-processing step to transform learned low-level control functions into a Fourier basis. It is then possible to abstract (by ignoring components with coefficients below a certain level) the control functions to yield a compact representation that is suitable for parameterization. The low-level control functions are open-loop controllers but they are incorporated into a high-level controller that is a function of the system state. This is done by simulating the effects of each possible low-level controller over some time interval and then picking the one that, in terms of the current goal, worked best. If simulation is computationally expensive and there are many low-level controllers then this is inefficient. It is interesting to note that for non-statically stable motions this approach cannot generally be used in a real-world controller. This is because in the real world we can not run time backwards to "undo" our mistakes.

Representation. In practice the control function and the state space trajectory are represented by a discrete set of basis functions over some finite interval of their domain. In [123] they are discretized as a finite set of samples. This can lead to problems of unwieldy dimensionality, so [32] split motions into different spacetime windows and constrained the motion to be representable by a spline basis function. The dimensionality of the problem is further and drastically reduced in [72] by using techniques from compiler optimization to factor out common subexpressions and by using a hierarchical wavelet basis function to represent the motion. This purportedly leads to smaller control problems, better numerical conditioning and faster convergence. For highly non-linear problems, however, difficulties may still remain in the numerical solution. The control function in [119] is a step function (called a "pose control graph") that can be viewed as a (cyclic) finite state machine with timed transitions between states.

Usability. The approach in [118] produces a family of optimal control solutions that can be reused and conveniently sequenced to produce complicated motion sequences. In [117, 87] the user is only required to supply a physical description of the object (including its sensors and actuators) and the algorithm will attempt to calculate "useful" modes of locomotion. In [102] even the creature morphology is generated automatically. While this may contain a degree of novelty it is hard to imagine a wide

range of situations in which an animator may be able to allow a creature's appearance to be chosen at random.

2.6 High-Level Requirements

Enforcing high-level constraints on character behavior is an example of a high-level requirement. The use of models to enforce such constraints within computer animation has, however, been somewhat uncommon. The seminal paper in this area was Reynolds [96], in which reactive behaviors, such as flocking behavior, were synthesized. Since then others have taken the approach to new levels of sophistication. In [114] the utility of autonomous agents for animating natural ecosystems was demonstrated. This approach is ideal, when compared with conventional "key-framing" approaches, for producing convincing "background" animations. In [23] a similar (kinematic) system is described that can potentially be used to bring autonomous agents more into the foreground. By allowing the user to interactively direct an agent they demonstrate their system to be a powerful tool for virtual reality.

Some of the high-level behaviors achieved in [114, 23] (such as "escaping", "mating", etc.) overlap with the capabilities of our system. However, the creatures in these systems have no "understanding" at all of what they are doing. Their behaviors are a direct result of the detailed rules programmed into them, and any interactive direction. Consequently, it is difficult to build, reconfigure or substantially extend the resulting behavior control systems. Moreover, while this previous research demonstrates how behavioral animation makes producing animations easier once the behaviors are defined, it does not address the issue of how to make *developing* such behaviors easier. For the low-level behaviors common to all creatures (such as "avoiding obstacles") this issue is less important because they need only be implemented once. However, high-level behaviors (such as "if you lost your sword, then hide from monsters") are what define the character. Consequently, every time we want a new character (with significantly different behavior) we will need to rewrite the high-level behaviors. Previous work implicitly assumes that any imperative computer programming language will suffice for this task. However, the computer program corresponding to the desired behavior will not generally be at all obvious. This is because a computer program is a specification of how we want the computer to behave. As such we may be forced to think in terms of machine state and assignment operators, instead of intuitively in terms of actions and effects. For example, it is simple to specify a sorted list but it took a great deal of ingenuity to go from this to the first implementation the "heap sort" algorithm. Even understanding how a given computer program implements a given specification can take a long time. Therefore there are many advantages to only concerning ourselves with high-level specifications. The penalty for such an approach will invariably be inefficiency, but, there are numerous occasions efficiency is not a primary concern. That is, with any computer program there is a trade off between the programmer's extra time spent developing an efficient program versus the user's extra time spent waiting for an inefficient program. For a one-off animation the trade-off may well be weighted in favor of reducing the programmer's effort. This is also the case in rapid prototyping.

There is an extensive and important body of relevant work in the intelligent autonomous agents literature [19, 121]. One piece worthy of particular mention is that described in [126]. In terms of motivation and results, this work has some overlap with the material we present in chapter 4. However, our work differs from all previous work in that it tries to be as "neat" as possible in dealing with the same set of problems. It may turn out that a completely formal and mathematical approach is not always appropriate. We simply want to try to push the formal approach as far as possible. If we are successful then we will have lost nothing but gained all the advantages associated with a high-degree of *clarity*.

Our work has much in common with work in the logic programming community [3], in which a more intuitive, declarative approach to specifying character behavior is used. In particular, we have been highly motivated by work that has been applied to computer animation [41, 9, 73]. Our work is different in that the semantics for our behavior specifications are given entirely by first-order logic. As far as we aware none of the work in this area, applicable to the problems we address (aside from that we shall discuss later), uses such simple semantics for their theories of action. We do not refute the need for more powerful logics. It is just that we want to try to use the simplest, best understood, and most widely known logic, that we perceive as being adequate, to solve as many problems as we can. To this end we have chosen to use a rigorous and

general first-order theory of action based on a version of the situation calculus [76]. The situation calculus is well-known, simple and intuitive to understand. The version we use is inspired by new work in cognitive robotics that solves the frame problem [94], incorporates knowledge-producing actions (like sensing) [98], and allows advice to be conveniently given as a "sketch plan" [68].

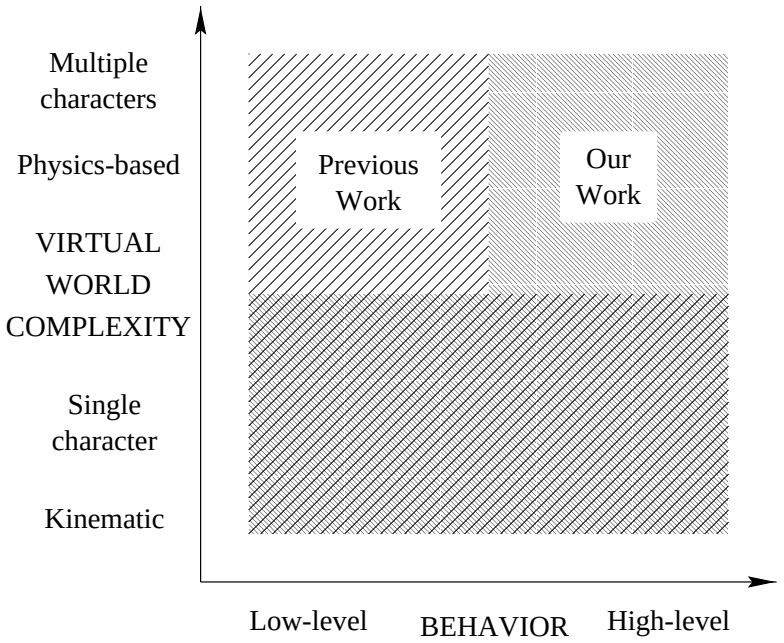


Figure 2.4: Design Space

Figure 2.4 attempts to show pictorially how our work extends previous work in the area. Like previous work our approach can handle low-level and high-level behavior in virtual worlds of low complexity. As we start to consider physics-based worlds or include multiple agents other approaches start to founder. Our work is able to venture into this area because we include a notion of sensing. This addition allows us to use the situation calculus beyond its widely perceived limitations to reason about dynamic worlds with multiple agents. We believe that in the future our work can also handle low-level behavior in unpredictable worlds. To be fully convincing, however, we will need to pay more attention to continuous actions.

2.7 Our Work

This chapter has given a clear picture of where our work fits in to computer animation research. In addition the reader will have a clear understanding of the foundations and terminology that we shall employ in what follows. It should also be clear that our work is a natural progression in the attempt to build computer models to satisfy ever more sophisticated requirements. In particular we would like to build cognitive models to help us automatically produce realistic behavior. These models will sit on top of biomechanical, physical and kinematic models. Moreover, the theory of action that we use to build our cognitive models is independent of the realization of the actions and of the underlying models. In chapter 4 the underlying models are kinematic, whereas in chapter 5 they are physics-based.

Communication between the different levels in the hierarchy is by virtue of actions. Actions to be performed are communicated to the lower levels where they are executed by the appropriate routines. Some actions are sensing actions. Their function is to prompt the lower levels to return pertinent information about the state of the world to the cognitive level. Because we use logic for our cognitive models it is not immediately obvious how to write down things about the lower levels about which we are uncertain. That is we must always write down things that are true but we are not precisely sure what is true. In previous work researchers have used modal logic approaches to write down facts about what the possible state of the

world is. We prefer instead to use intervals. Intervals give us a succinct and practical way to make concrete statements about quantities we may not know precisely. We shall discuss the details of the cognitive modeling language in the next chapter.

It is possible that there are examples where the clean stratification into cognitive and non-cognitive models is not appropriate. However, there is a large and useful body of problems where it is a useful abstraction. It is thus a sensible place to start and we shall focus on this paradigm for the remainder of the document. If and when it proves inadequate we can seek to further push our approach into this new more complicated arena.

Chapter 3

Theoretical Basis

The cornerstone of our approach to generating high-level behavior is to use the *situation calculus* to model the virtual world from the animated character's point of view. The situation calculus [76, 94] is a way of describing change in sorted first-order logic. In this chapter we shall give the mathematical details required to understand the situation calculus and how we use it. We necessarily assumes some familiarity with mathematical logic and the reader is referred to [37] for any additional background information required.

We shall use the following simple example, after [55], to illustrate various points about the situation calculus:

> Suppose we have two, call them Dognap and Jack, characters situated in some virtual world. The virtual world may be part of some computer game or perhaps the basis for some "virtual movie director" software. Regardless, let us suppose that Dognap is armed with a gun, and that Dognap wants to kill Jack. Let us further suppose that Jack is initially alive and that the gun is initially empty.

3.1 Sorts

A *situation*, of sort SITUATION, is a "snapshot" of the state of the world. A domain-independent constant s_0, of sort SITUATION, denotes the initial situation.

We want to use various number systems in our theory of action. Logical accounts of numbers are, however, problematic. Consequently, we wish to avoid giving such an account, and thus becoming embroiled in a task that would be at odds with the purpose of this chapter. That is, we want to remain focussed on building cognitive models for animated characters. The main artifices we shall use to allow us to eschew the messy details of arithmetic are as follows:

- A collection of sorts for various number systems:

$$\begin{aligned} \mathbb{B} &\triangleq \text{Boolean numbers,} \\ \mathbb{N} &\triangleq \text{Natural numbers,} \\ \mathbb{Z} &\triangleq \text{Integer numbers,} \\ \mathbb{Q} &\triangleq \text{Rational numbers,} \\ \mathbb{R} &\triangleq \text{Real numbers.} \end{aligned}$$

  To avoid misunderstanding, we briefly clarify $\mathbb{B}$.[1] In particular, there are two constants of sort $\mathbb{B}$, namely 0 and 1. There is one unary function $\neg$, and two binary functions $\wedge, \vee$.

  Later on, we shall want to ensure that all our number systems have maximal and minimal elements. We shall indicate these augmented number systems with a $\star$, for example the extended real numbers:

[1] See [57] for further detailed discussion on the subject.

$\mathbb{R}^\star = \mathbb{R} \cup \{-\infty, \infty\}$. We shall also denote subsets of our number systems with appropriate designators, for example the non-negative reals $\mathbb{R}^+$.

- For each number system sort, we will only consider standard interpretations. That is, we shall work with interpreted theories in which the various functions and constants, associated with the sort, are fixed. There are functions and predicates corresponding to all the standard mathematical functions and predicates for the sort.
- For each number system sort, we assume the existence of an "oracle" that is capable of determining the truth or falsity of sentences about relationships between objects of that sort.[2]

All other objects are of sort OBJECT.

3.2 Fluents

Any property of the world that can change over time is known as a fluent. A *fluent* is a function, with a situation term as (by convention) its last argument. We shall restrict fluents to taking on values in one of the number system sorts. For any functional fluents foo that take on values in $\mathbb{B}$ we shall adopt the standard abbreviation that $\mathsf{Foo}(s)$ is just shorthand for $\mathsf{foo}(s) = 1$. We may refer to such fluents as *relational fluents*.

Let us now introduce some fluents to capture the salient details of our example. This will enable us to formalize the scenario within the situation calculus.

$\mathsf{Alive}(s)$	–	Jack is alive in state s.
$\mathsf{Aimed}(s)$	–	The gun is aimed at Jack in state s.
$\mathsf{Loaded}(s)$	–	The gun is loaded in state s.

Actions, of sort ACTION, are the fundamental instrument of change in our ontology. The situation s' resulting from doing action a in situation s is given by the distinguished function $do : \text{ACTION} \times \text{SITUATION} \rightarrow \text{SITUATION}$, such that, $s' = do(a, s)$. In our example, we introduce the following actions:

load	–	Load the gun.
aim	–	Aim the gun at Jack.
shoot	–	Shoot the gun.

The possibility of performing action a in situation s is denoted by a distinguished predicate $Poss : \text{ACTION} \times \text{SITUATION}$. Sentences that specify what the state of the world must be before performing some action are known as *precondition axioms*. We can give such axioms for the actions in our example:[3]

$Poss(\mathsf{load}, s)$	–	The gun can always be loaded.
$Poss(\mathsf{aim}, s)$	–	The gun can always be aimed at Jack.
$Poss(\mathsf{shoot}, s) \Rightarrow \mathsf{Loaded}(s)$	–	The gun can only be shot if it's loaded.

3.3 The Qualification Problem

The *qualification problem* [75] is that of trying to infer when an action is possible. In our example, we only wrote down certain necessary conditions, we did not enumerate all the things that may prevent us from shooting the gun. For instance, we cannot shoot if the trigger is too stiff, or if the gun is encased in concrete, etc. By employing a *closed-world assumption*, we may obviate this problem and assume that our set of necessary conditions is also a sufficient set. For instance, under this assumption our precondition axiom for shoot now becomes:

$$Poss(\mathsf{shoot}, s) \Leftrightarrow \mathsf{Loaded}(s).$$

In general, we have the following definition:

[2] From a practical point of view, we might use mathematical software packages (such as Maple) to handle a wide range of useful queries.

[3] Throughout, all unbound variables are implicitly assumed to be universally quantified.

Definition 3.3.1 (Action precondition axioms). *Action precondition axioms give necessary and sufficient conditions $\pi_a(\vec{x}, s)$ for when an action $a(\vec{x})$ is possible. They are of the form:*

$$\mathit{Poss}\,(a(\vec{x}), s) \Leftrightarrow \pi_a(\vec{x}, s).$$

In [44, 70], some additional subtleties, including those that arise when we allow state constraints, are discussed.

3.4 Effect Axioms

Effect axioms give necessary conditions for a fluent to take on a given value after performing an action. We can use effect axioms to state the effects of the actions on the defined fluents in our example:

$\mathsf{Loaded}(\mathit{do}(\mathsf{load}, s))$	–	The gun is loaded after loading it.
$\mathsf{Aimed}(\mathit{do}(\mathsf{aim}, s))$	–	The gun is aimed at Jack after aiming it.
$\mathit{Poss}\,(\mathsf{shoot}, s) \wedge \mathsf{Aimed}(s)$	$\Rightarrow$	$\neg\mathsf{Alive}(\mathit{do}(\mathsf{shoot}, s))$
	–	If the gun is aimed at Jack and it can be shot then he is dead after shooting it.

All that now remains to complete our first pass at formalizing our example is to specify the initial situation:

$\mathsf{Alive}(\mathrm{S}_0)$	–	Initially Jack is alive.
$\neg\mathsf{Aimed}(\mathrm{S}_0)$	–	Initially the gun is not aimed at Jack.
$\neg\mathsf{Loaded}(\mathrm{S}_0)$	–	Initially the gun is not loaded.

3.5 The Frame Problem

Unfortunately, there are still some impediments to using the situation calculus in real applications. The most notable of these is the so called *frame problem* [76]. The frame problem is that of trying to infer what remains unchanged by an action. In our example, we only wrote down what changed after an action; we did not write down all the things that stayed the same. For instance, the gun stayed loaded after aiming it, or the gun did not turn into a horse after loading it, etc. In common-sense reasoning about actions, it seems essential to assume that, unless explicitly told otherwise, things stay the same. To formally state this "law of inertia", without changing our effect axioms, causes problems. In particular, if we have $\mathcal{A}$ actions and $\mathcal{F}$ fluents, then we must write down a set of $\mathcal{A} \times \mathcal{F}$ "frame" axioms. The problem is exacerbated by the planner having to reason efficiently in the presence of all these axioms.

At this point it is worth commenting on how the frame problem is dealt with in other fields that tackle related problems. Notably in control theory. In control theory we have the notion of a state vector. Each component of the state vector is similar to a fluent. The frame problem is tackled by simply assuming that the state vector completely characterizes the system in question and that values for all the components are explicitly specified. That is, if part of the state vector is unchanged then we must explicitly say so. Usually state vectors are chosen to be short so that this task is not too irksome. It does, however, put our approach into context. We do not want to be constrained to have to give our state vector at the outset. Moreover, we do not want (for reasons given in the preceeding paragraph) to list out all the things that don't change. In our approach we can, at any point, mention some new fluent and have the system infer its value with respect to its value in the initial situation. Furthermore, if so desired, we can leave the initial situation underspecified. For example, suppose in the initial situation we say that the car is *either* blue *or* yellow. Now further suppose we perform no actions to affect the color. Then, after the action sequence, we will be able to infer that the car's color is still either blue, or yellow.

In [94], it is shown how we can avoid having to list out all the frame axioms. The idea is to assume that our effect axioms enumerate all the possible ways that the world can change. This closed world assumption provides the justification for replacing the effect axioms with *successor state* axioms. For instance, the

successor state axiom for $\mathsf{Alive}(s)$ states that Jack is alive, if and only if, he was alive in the previous state and he was not just shot:

$$\mathit{Poss}(a, s) \Rightarrow [\mathsf{Alive}(\mathit{do}(a, s)) \Leftrightarrow \mathsf{Alive}(s) \wedge \neg(a = \mathsf{shoot} \wedge \mathsf{Aimed}(s))]. \tag{3.1}$$

In general, we have the following definition:

Definition 3.5.1 (Successor state axioms). *Suppose $\gamma_f(\vec{y}, z, a, s)$ is a first-order formula whose free variables are among $\vec{y}, z, a, s$. Assume it states all the necessary conditions under which action a, if performed in s, results in $f(\vec{y}, s)$ becoming equal to z. Then, the corresponding successor state axiom, that assumes the given conditions are also sufficient ones, is of the form:*

$$\mathit{Poss}(a, s) \;\Rightarrow\; [(f(\vec{y}, \mathit{do}(a, s)) \;=\; z) \;\Leftrightarrow\; (\gamma_f(\vec{y}, z, a, s)) \vee (f(\vec{y}, s) \;=\; z \wedge \neg\exists z' \;\; \gamma_f(\vec{y}, z', a, s))]. \tag{3.2}$$

It is instructive to consider what this definition means for a relational fluent F. Let $\gamma_F^+(\vec{y}, a, s)$ be a disjunction of all the positive effects of the action a, and $\gamma_F^-(\vec{y}, a, s)$ be a disjunction of all the negative effects. Then the successor state axiom for F is:

$$\mathit{Poss}(a, s) \Rightarrow [F(\vec{y}, \mathit{do}(a, s)) \Leftrightarrow (\gamma_F^+(\vec{y}, a, s) \vee (F(\vec{y}, s) \wedge \neg\gamma_F^-(\vec{y}, a, s)))].$$

3.5.1 The Ramification Problem

Suppose we add to our example the fluents

$\mathsf{NearBomb}(s)$ – Jack is near the position of the bomb in state s;
$\mathsf{Fire}(s)$ – There is a fire in state s;

the action

$\mathsf{detonate}$ – The bomb is detonated;

the effect axiom (assuming it is always possible to detonate the bomb):

$\mathsf{Fire}(\mathit{do}(\mathsf{detonate}, s))$ – There is a fire after detonation;

and the state constraint

$\mathsf{Fire}(s) \wedge \mathsf{NearBomb}(s) \Rightarrow \neg\mathsf{Alive}(s)$
– If Jack is near a fire, then he is dead.

Then detonating the bomb, when Jack is next to it, has the implicit side effect of killing Jack. Such side-effects are known as ramifications. So the *ramification problem* is that of trying to determine all the implicit side-effects of actions caused by the presence of state constraints. In general, state constraints may give rise to intractable problems (see [70]). However, in some cases, we can deal with the ramification problem by making all the side effects explicit. This can be done automatically (see [70]) by "compiling" the state constraint into the successor state axioms. For example, we can modify the successor state axiom for $\mathsf{Alive}(s)$ as follows:

$$\mathit{Poss}(a, s) \Rightarrow [\mathsf{Alive}(\mathit{do}(a, s)) \;\Leftrightarrow\; \mathsf{Alive}(s) \wedge \neg(a = \mathsf{shoot} \wedge \mathsf{Aimed}(s)) \wedge \neg(a = \mathsf{detonate} \wedge \mathsf{NearBomb}(s)).]$$

3.6 Complex Actions

A human actor receives advice on how to behave from a director. We would like to develop an analogous approach for directing synthetic actors. The required notion of an advice taker as a useful tool for computer science was first proposed in 1963 by John McCarthy [74]. It provided the original motivation for the

situation calculus, but a realistic proposal for incorporating advice has only recently been developed [68]. In this section, we describe how this approach can enable a character to intelligently follow an animator's high-level advice. In particular, we describe the language used to give these instructions and show how the character can use its background knowledge to fill in the details that the animator chooses not to specify.

The actions we discussed previously, defined by corresponding precondition and successor state axioms, are referred to as a *primitive actions*. We have explained how they can be used by the character to keep track of its changing world. Until now, however, we have not mentioned where the actions come from in the first place. This is the issue to which we now turn our attention. In particular, by using the notion of macros, [67, 66] show how to define new *complex actions* in terms of the previously defined primitive actions. For example, we may want to add a complex action like:

$$\textbf{if } \mathsf{haveExtinguisher} \textbf{ then } \mathsf{detonate} \textbf{ else } \mathsf{shoot}.$$

Here the state argument of fluents in complex actions is suppressed. It is re-inserted as the macro is expanded into the corresponding situation calculus term, see appendix D for details.

Complex actions correspond to our previous informal notion of a "sketch plan". The effect of a complex action α is defined by the macro $\mathit{Do}(\alpha, s, s')$, where s' is a state that results from doing α in state s. These *macros* expand out into situation calculus expressions, thus ensuring complex actions inherit the solution to the frame problem for primitive actions. Given some advice, represented as a complex action $\mathsf{program}$, the underlying theorem prover (reasoning engine) in the character's reasoning system attempts to prove:

$$\textsc{Axioms} \models \exists s\ \mathit{Do}(\mathsf{program}, \mathrm{S}_0, s).$$

The resulting (constructive) proof results in a term $s = \mathit{do}(a_n, \cdots, \mathit{do}(a_1, \mathrm{S}_0))$, such that $[a_1, \ldots, a_n]$ is the sequence of primitive actions that the character should perform in order to follow our advice.

The complete list of operators for defining the complex actions that we use is given below. Together they define the *high-level control language* used for issuing advice to characters. The mathematical definitions for these operators is given in appendix D.

(Sequence) $\alpha \fatsemi \beta$ means do action α, followed by action β;

(Test) $p?$ means do nothing if p is true, otherwise fail;

(Conditionals) **if** p **then** α **else** β, means do α if p is true, otherwise do β;

(Non-deterministic iteration) $\alpha\star$, means do α zero or more times.

(Iteration) **while** p **do** α **od**, means do α while p is true;

(Nondeterministic choice of actions) $\alpha|\beta$ means do action α, or action β;

(Nondeterministic choice of arguments) $(\pi x)\alpha(x)$ means pick some argument x and perform the action $\alpha(x)$;

(Procedures) **proc** $P(x_1, \ldots, x_n)\alpha$ **end** declares a procedure $P(x_1, \ldots, x_n)$.

A key feature to notice is the ability to use nondeterminism. This allows us to, potentially, write extremely high-level specifications of behavior. For example, at one extreme, we have the classic planning problem represented by the "program":

$$\begin{array}{l}\textbf{while } \neg\mathsf{Goal}\\ \quad (\pi\ a)[\mathsf{Appropriate}(a)? \fatsemi a]\\ \textbf{od}.\end{array}$$

At the other extreme we can choose to stick to laboriously specify every detail as in conventional programming. The middle ground between these two alternatives makes for a useful and novel methodology for writing controllers.

For example, the following "program" illustrates some of the power of the approach.

proc kill
  detonate | (shoot) ⋆
end

kill ⨟ arrangeFuneral

The kill procedure is defined to instruct Dognap to detonate the bomb or to shoot zero or more times. Imagine that Jack first considers detonating the bomb. One of the effects of the detonation is that Dognap looses all his money in compensation claims. Thus he would be unable to pay for the funeral. That is, we suppose being rich to be a precondition of the arrangeFuneral action. This will cause the interpreter to backtrack to search for alternatives. Next Dognap considers shooting Jack. Unfortunately, the effect of shooting Jack once is only to wound. There can be no funeral without a corpse and so the interpreter back tracks again. It once again rejects an explosion and moves on to consider shooting twice. This time Jack will be dead and Dognap can still afford the funeral costs. Therefore the interpreter selects this course of action as the right one. Moreover, the other conditions can be specified concisely and, thanks to the above solution to the frame problem, perspicuously.

The possibility also exists for incremental refinement of a specification, perhaps, from a high-level specification to the point where it more closely resembles a controller written using a conventional imperative programming language.

In appendix E we discuss some issues that surround implementing an interpreter for complex actions.

3.7 Exogenous Actions

It will often be the case that there are aspects of the domain that we can not, or do not want to, formalize. For example, suppose we are interested in the position of a ball floating in the (virtual) sea. It is somewhat irrelevant to attempt to formalize the motion of the waves, wind, etc. This will often be the case for phenomena that are outside the characters ability to control. We should like to simply define an action like moveBall($\boldsymbol{x}$) and say that it is caused by mysterious external forces. Such actions are referred to as *exogenous* actions. While the cause of an exogenous action is difficult to state its effect need not be. For example, the moveBall($\boldsymbol{x}$) simply moves the ball to the position $\boldsymbol{x}$.

Exogenous actions can be incorporated into the situation calculus by modifying the definition of the macro expansion for complex actions to allow for the possible occurrence of exogenous actions. We shall see an example of this in section 5.4.2 where we consider physics-based virtual worlds.

3.8 Knowledge producing actions

Up until now we have thought of actions as having effects on the world. We can, however, imagine actions whose only effect is to change what the character knows about its world. A good example is a character trying to make a phone call. The character needs to know the number before dialing. The action of looking up the phone number has no effect on the world, but it changes the character's knowledge. Sensing actions are therefore referred to as *knowledge producing actions*.

In [98], an approach to incorporating knowledge producing actions into the situation calculus is described. The idea behind the approach is to define an epistemic fluent to keep track of all the worlds a character thinks it might possibly be in. In chapter 1, figure 1.2 depicted a character unable to decide which world it was in. That is, whether in its world the light was on or off. Figure 3.1 shows the character turning around to see that the light is in fact turned on. The result of this sensing action is shown in the figure as the character discarding some of the worlds it previously thought were possible. In particular, since it now knows that the light is on in its world, it must throw out all the worlds in which it thought the light was

turned off. In this section we give the mathematical details of how this notion is modeled in the situation calculus.

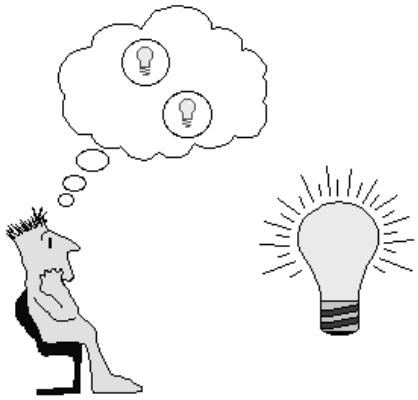


Figure 3.1: After sensing, only worlds where the light is on are possible.

3.8.1 An epistemic fluent

The way a character keeps track of the possible worlds or, as the case may be, possible situations is to define an epistemic fluent K. The fluent keeps track of all the K-related worlds. These K-related worlds are precisely the ones in the bubbles above the characters head in above mentioned figures. They are the situations that the character thinks might be its current situation. So we write $\mathsf{K}(s', s)$ to mean that in situation s, as far as the character can tell, it might be in the alternative situation s'. That is, the character's knowledge is such that s and s' are indistinguishable. It can only find out if it is or not by sensing the value of certain terms, for example terms such as $\mathsf{light}(s)$.

When we say a character *knows* the value of a term τ, in a situation s, is some constant c, we mean that τ has the value c in all the K-related worlds. For convenience, we introduce the following abbreviation:

$$\mathit{Knows}(\tau = c, s) \triangleq \forall s' \ \mathsf{K}(s', s) \Rightarrow \tau[s'] = c, \tag{3.3}$$

where $\tau[s']$ is the term τ with the situation arguments inserted. For example, if $\tau = \mathsf{phoneNo}(\mathsf{Jack})$ then $\tau[s] = \mathsf{phoneNo}(\mathsf{Jack}, s)$. Note that for simplicity we are considering the case where we only have one character. For more than one character we simply need to make it clear which character knows what. For example, $\mathit{Knows}(\mathsf{Dognap}, \tau = c, s)$ indicates that Dognap knows the value of τ.

When a character knows the value of a term, but we do not necessarily know the value of the term, we use the notation $\mathit{Kref}(\tau, s)$ to say that the character *knows the referent* of τ:

$$\mathit{Kref}(\tau, s) \triangleq \exists z \ \mathit{Knows}(\tau = z, s). \tag{3.4}$$

We now introduce some special notation for the case when τ takes on values in $\mathbb{B}$. In particular, since there are only two possibilities for the referent, we say we *know whether* τ is true or not:

$$\mathit{Kwhether}(\tau, s) \triangleq \mathit{Knows}(\tau = 1, s) \vee \mathit{Knows}(\tau = 0, s). \tag{3.5}$$

3.8.2 Sensing

As in [98], we shall make the simplifying assumption that for each term τ, whose value we are interested in sensing, we have a corresponding knowledge producing action sense_τ. In general, if there are n knowledge

producing actions: sense_{τ_i}, $i = 0, \ldots, n-1$, then we shall assume there are n associated situation dependent terms: $\tau_0, \ldots, \tau_{n-1}$. The corresponding successor state axiom for K is then:

$$\begin{aligned}\mathit{Poss}(a, s) \Rightarrow [\mathsf{K}(s'', \mathit{do}(a, s)) \Leftrightarrow \\ \exists s' \ (\mathsf{K}(s', s) \wedge (s'' = \mathit{do}(a, s'))) \wedge \\ ((a \neq \mathsf{sense}_{\tau_0} \wedge \cdots \wedge a \neq \mathsf{sense}_{\tau_{n-1}}) \\ \vee (a = \mathsf{sense}_{\tau_0} \wedge \tau_0(s') = \tau_0(s)) \\ \vdots \\ \vee (a = \mathsf{sense}_{\tau_{n-1}} \wedge \tau_{n-1}(s') = \tau_{n-1}(s)))]. \quad (3.6)\end{aligned}$$

The above successor state axiom captures the required notion of sensing and solves the frame problem for knowledge producing actions. We shall explain how it works through a simple example. In particular, let us consider the problem of sensing the current temperature. Firstly, we introduce a fluent $\mathsf{temp} : \textsc{situation} \rightarrow \mathbb{R}^+$, that corresponds to the temperature (in Kelvin) in the current situation. For now let us assume that the temperature remains constant:

$$\mathit{Poss}(a, s) \Rightarrow \mathsf{temp}(\mathit{do}(a, s)) = \mathsf{temp}(s). \quad (3.7)$$

We will have a single knowledge producing action $\mathsf{senseTemp}$. This gives us the following successor-state axiom for K:

$$\begin{aligned}\mathit{Poss}(a, s) \Rightarrow [\mathsf{K}(s'', \mathit{do}(a, s)) \Leftrightarrow \exists s' \ (\mathsf{K}(s', s) \wedge (s'' = \mathit{do}(a, s'))) \wedge \\ ((a \neq \mathsf{senseTemp}) \vee (a = \mathsf{senseTemp} \wedge \mathsf{temp}(s') = \mathsf{temp}(s)))]. \quad (3.8)\end{aligned}$$

The above axiom states that for any action other than $\mathsf{senseTemp}$ the set of K-related worlds is the set of images of the previous set of K-related worlds. That is, if s' was K-related to s, then the image $s'' = \mathit{do}(a, s')$, of s' after performing the action a is K-related to $\mathit{do}(a, s)$. Moreover, when the character performs a $\mathsf{senseTemp}$ action, in some situation s, the effect is to restrict the set of K-related worlds to those in which the temperature agrees with the temperature in the situation s. In other words, $\mathsf{senseTemp}$ is the only knowledge producing action, and its effect is to make the temperature denotation known: $\mathit{Kref}(\mathsf{temp}, \mathit{do}(\mathsf{senseTemp}, s))$. The reader is referred to [98] for any additional details, examples or theorems on any of the above.

3.8.3 Discussion

The formalization of knowledge within the situation calculus using the epistemic fluent K makes for an elegant mathematical specification language. It is also powerful. For example, suppose we have an effect axiom that states that if a gun is loaded then the character is dead after shooting the gun:

$$\mathsf{Loaded}(s) \Rightarrow \mathsf{Dead}(\mathit{do}(\mathsf{shoot}, s)).$$

Furthermore, suppose we know the gun is initially loaded $\mathit{Knows}(\mathsf{Loaded}, \mathrm{s}_0)$, then we can infer that we know the character is dead after shooting the gun $\mathit{Knows}(\mathsf{Dead}(\mathit{do}(\mathsf{shoot}, \mathrm{s}_0))$.

Unfortunately, there are some problems. One set of problems is associated with implementation, the second applies to reasoning about real numbers, both in theory and in practice.

Implementation

The implementation problems revolve around how to specify the initial situation. For example, if we choose an implementation language like Prolog, specifying the initial situation may involve having to list out an exponential number of possible worlds. For example, if we do not initially know if the gun is loaded then we might consider explicitly listing the two possible worlds `s_a`, and `s_b`, such that:

```
k(s_a,s0).
k(s_b,s0).
loaded(s_a).
```

As we add more relational fluents, that we want to be able to refer to our knowledge of, the situation gets worse. In general, if we have n such fluents, there will be 2^n initial possible worlds that we have to list out. Once we start using functional fluents, however, things get even worse: we cannot, by definition, list out the uncountably many possible worlds associated with not knowing the value of a fluent that takes on values in $\mathbb{R}$.

Intuitively, we need to be able to specify rules that characterize, without having to list them all out, the set of initial possible worlds. It may be possible to somehow coerce Prolog into such an achievement. Perhaps, more reasonably, we could consider using a full first-order logic theorem prover. However, first-order logic theorem provers are inefficient and experimental. In addition, in section 1.5 we advance the idea that our approach can be used for rapid prototyping. This claim relies on the possibility of gradually removing the non-determinism from our specifications. In this way we might hope to eventually refine a specification so that it can be run without the need for an underlying theorem prover. This idea must, sadly, be forsaken if we are to ingrain the need for a theorem prover into our approach to sensing.

Ignoring all the above concerns let us assume that we can specify rules that characterize the set of initial possible worlds. For example, suppose that initially we know the temperature is between 10 and 50 Kelvin. We might express this using inequalities:

$$\forall s' \ \mathsf{K}(s', \mathrm{s}_0) \Rightarrow 10 \leqslant \mathsf{temp}(s') \leqslant 50.$$

This, however, brings us to our second set of problems related to reasoning about real numbers.

Real numbers

We just wrote down the formula that corresponds to:

$$\mathit{Knows}(10 \leqslant \mathsf{temp} \leqslant 50, \mathrm{s}_0). \tag{3.9}$$

Suppose, we are now interested in what this tells us about what we know about the value of the temperature squared. In general, if we know a term τ lies in the range $[u, v]$ we would like to be able to answer questions about what we know about some arbitrary function f of τ. Such questions take us into a mathematical minefield of reasoning about inequalities. Fortunately, a path through this minefield has already been charted by the field of interval arithmetic.

3.9 Interval arithmetic

To address the issues we raised in section 3.8.3 we turn our attention to interval arithmetic [82, 83, 115]. Some of the immediate advantages interval arithmetic affords us are listed below:

- Interval arithmetic enables us to move all the details of reasoning about inequalities into the rules for combining intervals under various mathematical operations.
- Interval arithmetic provides a finite (and succinct) way to represent uncertainty about a large, possibly uncountable, set of alternatives. Moreover, the representation remains finite after performing a series of operations of the intervals. In [89] interval arithmetic is compared to probability as a means of representing uncertainty.
- Writing a sound oracle for answering ground queries about interval arithmetic is a trivial task. Moreover, we can answer queries in time that is linear in the length of the query. Returning valid and optimal intervals is more challenging (see section 3.12). This should, however, be compared to the vastly unrealistic assumption we (and others) made earlier about the existence of oracles for answering queries about the real numbers.

- There is no discrepancy between the underlying theory of interval arithmetic, and the corresponding implementation. Thus we re-establish our claims about using our approach for rapid prototyping.

We construct interval arithmetics from our previously available number systems as follows:

- For each number system $\mathbb{X}$, we add a new number system sort $\mathcal{I}_{\mathbb{X}}$. The constants of $\mathcal{I}_{\mathbb{X}}$ are the set of pairs $\langle u, v\rangle$ such that $u, v \in \mathbb{X}$ and $u \leqslant v$. There are functions and predicates corresponding to all the functions and predicates of $\mathbb{X}$.
- For an interval $\boldsymbol{x} = \langle u, v\rangle$, we use the notation $\underline{\boldsymbol{x}} = u$ for the lower bound, and $\overline{\boldsymbol{x}} = v$ for the upper bound.
- The function width, returns the width of an interval $\boldsymbol{x}$, i.e. $\mathsf{width}(\boldsymbol{x}) = \overline{\boldsymbol{x}} - \underline{\boldsymbol{x}}$.
- When we have a number x and an interval $\boldsymbol{x} = \langle u, v\rangle$, such that $u \leqslant x \leqslant v$ we say that $\boldsymbol{x}$ contains x, we write $x \in \boldsymbol{x}$. Similarly for two intervals $\boldsymbol{x}$, $\boldsymbol{y}$ such that $\underline{\boldsymbol{y}} \leqslant \underline{\boldsymbol{x}}$ and $\overline{\boldsymbol{x}} \leqslant \overline{\boldsymbol{y}}$, we say that $\boldsymbol{y}$ contains $\boldsymbol{x}$, we write $\boldsymbol{x} \subseteq \boldsymbol{y}$.
- For two intervals $\boldsymbol{x}_0$, $\boldsymbol{x}_1$ we say that $\boldsymbol{x}_0 \leq \boldsymbol{x}_1$ if and only if $\overline{\boldsymbol{x}}_0 \leqslant \underline{\boldsymbol{x}}_1$.
- We let $\bot$ and $\top$ represent, respectively, the minimum and maximum elements of the number system in question. For example, in $\mathbb{R}^\star$, $\langle \bot, \top\rangle = \langle -\infty, \infty\rangle$.

For example, consider the case of the number system $\mathcal{I}_{\mathbb{B}}$. There are three numbers in the number system: $\langle 0, 0\rangle$, $\langle 0, 1\rangle$ and $\langle 1, 1\rangle$. Note that we have $\langle 0, 0\rangle \leqslant \langle 0, 1\rangle \leqslant \langle 1, 1\rangle$, $\langle 0, 0\rangle \subset \langle 0, 1\rangle$, and $\langle 1, 1\rangle \subset \langle 0, 1\rangle$. In $\mathbb{B}$, 1 and 0 can be used to represent, respectively, "true" and "false". Similarly, $\langle 1, 1\rangle$, $\langle 0, 1\rangle$ and $\langle 0, 0\rangle$ in $\mathcal{I}_{\mathbb{B}}$ can be used to represent, respectively, "known to be true", "unknown", and "known to be false". We will discuss the details of how this is done in section 3.10.

By way of analogy, complex numbers are also made up of a pair of (real) numbers, and operations on them are defined in terms of operations on the reals. However, it would lead to confusion, if when reading a text on complex analysis we could not comprehend complex numbers as a separate entity, distinct from pairs of real numbers. We therefore forewarn the reader against making the same mistake for intervals. That is, although numbers in $\mathcal{I}_{\mathbb{X}}$ are made up of a pair of numbers from $\mathbb{X}$ it is important to treat them as "first-class" numbers in their own right.

Traditionally, interval arithmetic was used to address the innumerous problems with the ability of floating point arithmetic to accurately represent real arithmetic. For example, consider the real number $\sqrt{2}$. This real number cannot be represented exactly by any finite decimal. However, it can be represented by the *exact* interval $\langle 1.41, 1.42\rangle$. What this interval can be used to express is that the $\sqrt{2}$ lies somewhere between 1.41 and 1.42. That is, it expresses our uncertainty about the exact value of the $\sqrt{2}$ when expressed as a decimal. With modern computers our degree of uncertainty can be made miniscule and this is part of the appeal of interval arithmetic. Of course, the other part of interval arithmetic has to do with the arithmetic of intervals. We shall, however, delay any such discussion until section 3.12.

3.10 Interval-valued fluents

The epistemic K-fluent that we discussed previously allowed us to express a character's uncertainty about the value of a fluent in its world. Unfortunately, in section 3.8.3 we saw there were implementation problems associated with trying to represent a character's knowledge of the initial situation. Fortunately, in the previous section we saw that intervals also allow us to express uncertainty about a quantity. Moreover, they allow us to do so in a way that circumvents the problem of how to represent infinite quantities with a finite number of bits. It is, therefore, natural to ask whether we can also use intervals to replace the troublesome epistemic K-fluent.

The answer, as we shall seek to demonstrate in the remainder of this chapter, is a resounding "yes". In particular, we shall introduce new epistemic fluents that will be interval-valued. They will be used to represent a character's uncertainty about the value of certain non-epistemic fluents.

We have previously used functional fluents that take on values in any of the number systems: $\mathbb{B}$, $\mathbb{R}$, etc. There is nothing noteworthy about now allowing fluents that take on values in any of the interval numbers systems: $\mathcal{I}_{\mathbb{B}}$, $\mathcal{I}_{\mathbb{R}}$. Firstly, let us distinguish those regular fluents whose value maybe learned through a knowledge-producing action. We term such fluents *sensory fluents*. Now, for each sensory fluent f, we introduce a new corresponding interval-valued epistemic (IVE) fluent $\mathcal{I}_f$.

For example, we can introduce an IVE fluent $\mathcal{I}_{\mathsf{temp}}$: SITUATION $\rightarrow \mathcal{I}_{\mathbb{R}^{\star+}}$. We can now use the interval $\mathcal{I}_{\mathsf{temp}}(\mathrm{s}_0) = \langle 10, 50\rangle$ to state that the temperature is initially between 10 and 50 Kelvin. Similarly, we can even specify that the temperature is initially completely unknown: $\mathcal{I}_{\mathsf{temp}}(\mathrm{s}_0) = \langle 0, \infty\rangle$.

Our ultimate aim is that in an implementation we can use IVE fluents to completely replace the troublesome K-fluent. Nevertheless, within our mathematical theory, there is nothing to prevent our IVE fluents co-existing with our previous sole epistemic K-fluent. Indeed, if we define everything correctly then there are many important relationships that should hold between the two. These relationships take the form of state constraints and, as we shall show, can be used to express the notion of validity and optimality of our IVE fluents. If these state constraints are maintained as actions are performed then the IVE fluents completely subsume the troublesome K-fluent. This will turn out to be true until we consider knowledge of general terms. In which case we can maintain validity but may have to sacrifice our original notion of optimality (see section 3.13).

Seeking to make IVE fluent ubiquitous necessitates an alternative definition for *Knows* that does not mention the K-fluent. To this end, we introduce a new abbreviation, $\mathcal{I}_{\mathit{Knows}}$ such that for any term τ, $\mathcal{I}_{\mathit{Knows}}(\tau, s) = \langle u, v\rangle$ means that τ's *interval value* is $\langle u, v\rangle$. By "interval value" we mean the value we get by evaluating the expression according the set of rules that we shall discuss in section 3.13. For now, let us just consider the case when τ is some fluent f. When f is a sensory fluent then $\mathcal{I}_{\mathit{Knows}}$ is the value of the corresponding IVE fluent, otherwise it is completely unknown:

$$\mathcal{I}_{\mathit{Knows}}(f, s) = \begin{cases} \mathcal{I}_f(s) & \text{if } f \text{ is a sensory fluent,} \\ \langle \bot, \top\rangle & \text{otherwise.} \end{cases} \tag{3.10}$$

We now take the important step of redefining *Knows* to be the special case when $\mathcal{I}_{\mathit{Knows}}(\tau, s)$ has collapsed to a thin interval:

$$\mathit{Knows}'(\tau = c, s) \Leftrightarrow \mathcal{I}_{\mathit{Knows}}(\tau, s) = \langle c, c\rangle. \tag{3.11}$$

The definitions of *Kref*, and *Kwhether* are now in terms of the new definition for *Knows*′. As required, this new definition does not involve the problematic epistemic K-fluent.

We are now in a position to define what it means for an IVE fluent to be valid:

Definition 3.10.1 (Validity). *For every sensory fluent f, we say that the corresponding IVE fluent $\mathcal{I}_f$ is a* valid *interval if f's value in all of the* K*-related situations is contained within it:*

$$\forall s, s'\ \mathsf{K}(s', s) \Rightarrow f(s') \in \mathcal{I}_f(s).$$

Note that since we have a logic of knowledge (as opposed to belief) we have that every situation is K-related to itself: $\forall s\ \mathsf{K}(s, s)$. Thus, as an immediate consequence of definition 3.10.1, we have that if an IVE fluent $\mathcal{I}_f$ is valid then it contains the value of f: $\forall s\ f(s) \in \mathcal{I}_f(s)$.

The validity criterion is a state constraint that ensures the interval value of the IVE fluents is wide enough to contain all the possible values of the sensory fluents. It does not however prevent intervals from being excessively wide. For example, the interval $\langle -\infty, \infty\rangle$ is a valid interval for any IVE fluent that takes on values in $\mathcal{I}_{\mathbb{R}^{\star}}$. The notion of narrow intervals is captured in the definition of optimality:

Definition 3.10.2 (Optimality). *A valid IVE fluent $\mathcal{I}_f$ is also* optimal *if it is the smallest valid interval:*

$$\forall\ \boldsymbol{y}, s, s'\ \mathsf{K}(s', s) \Rightarrow (f(s') \in \boldsymbol{y} \Rightarrow \mathcal{I}_f(s) \subseteq \boldsymbol{y}).$$

3.11 Correctness

In this section we shall consider some of the consequences and applications of interval-valued fluents to formalizing sensing under various different assumptions. Our goal will be to show that we can maintain valid and optimal intervals as we perform actions. The first step will be to define successor state axioms for IVE fluents. This is done in much the same way as it was for regular fluents. For example, suppose we have a perfect sensor, then the following successor-state axiom states that after sensing, we "know" the temperature in the resulting situation

$$\mathit{Poss}(a,s) \Rightarrow [\mathcal{I}_{\mathsf{temp}}(\mathit{do}(a,s)) = \boldsymbol{y} \Leftrightarrow (a = \mathsf{senseTemp} \wedge \overline{\boldsymbol{y}} = \underline{\boldsymbol{y}} = \mathsf{temp}(s)) \vee (a \neq \mathsf{senseTemp} \wedge \mathcal{I}_{\mathsf{temp}}(s) = \boldsymbol{y})]. \quad (3.12)$$

Now let us consider the case in general. Firstly, we note that there is always an initial valid IVE fluent.

Lemma 3.11.1. *For any initial situation* s_0 *and sensory fluent* f *we have that* $\mathcal{I}_f = \langle \bot, \top \rangle$ *is a valid interval.*

Proof. The proof of the theorem is immediate from the fact that, by definition, $\langle \bot, \top \rangle$ bounds any possible value for f. So in particular it bounds all the values f can take in all the initial K-related situations. □

It is also the case that there will usually be an initial optimal interval.

Lemma 3.11.2. *If the initial set of* K*-related situations is either completely unspecified or specified with maximally restrictive inequalities then we can find an initial optimal IVE fluent for each of the sensory fluents.*

Proof. **Case (i)** The initial set of K-related situations is completely unspecified. That is, we are initially completely ignorant of a sensory fluent f's value. Then, the maximal interval is also clearly optimal. That is, $\langle \bot, \top \rangle$ is the *only* interval that bounds all possible values for f in the initial K-related situations. Since it is the unique valid interval it must, by definition, be an optimal interval.

Case (ii) We have a specification such as

$$(\forall s' \ \mathsf{K}(s', \mathrm{s}_0) \Rightarrow u \leqslant f(s') \leqslant v) \wedge$$
$$\neg\exists \ u', v' \ [u < u' \wedge v' < v \wedge (\forall s' \ \mathsf{K}(s', \mathrm{s}_0) \Rightarrow u' \leqslant f(s') \leqslant v')]$$

Then, consider $\mathcal{I}_f(\mathrm{s}_0) = \langle u, v \rangle$. As required, this is clearly the smallest valid interval.

□

In what follows we make the three following assumptions about all sensory fluents f:

1. The value of $\mathcal{I}_f$, in the initial situation, is optimal and valid. This assumption is justified by lemma 3.11.1 and 3.11.2.
2. The successor-state axiom for f is such that f remains constant:
$$\mathit{Poss}(a,s) \Rightarrow [f(\mathit{do}(a,s)) = f(s)]. \quad (3.13)$$
3. The successor-state axioms for each of the corresponding IVE fluents $\mathcal{I}_f$ are of the form:
$$\mathit{Poss}(a,s) \Rightarrow [\mathcal{I}_f(\mathit{do}(a,s)) = \boldsymbol{y} \Leftrightarrow (a = \mathsf{sense}_f \wedge \overline{\boldsymbol{y}} = \underline{\boldsymbol{y}} = \mathsf{f}(s)) \vee (a \neq \mathsf{sense}_f \wedge \mathcal{I}_f(s) = \boldsymbol{y})]. \quad (3.14)$$

We can now state our main correctness result.

Theorem 3.11.1. *With the above assumptions, for all situations s, and sensory fluents f, every IVE fluent $\mathcal{I}_f$ is valid and optimal.*

Proof. We shall prove the result by induction on s. We note that the base case follows by assumption 1. Therefore, we need only consider the case when $s^\star = \mathsf{do}(a, s)$.

By induction we may assume that

$$\forall s' \ \mathsf{K}(s', s) \Rightarrow f(s') \in \mathcal{I}_f(s),$$

and that $\mathcal{I}_f(s)$ is optimal. We seek to prove that

$$\forall s'' \ \mathsf{K}(s'', s^\star) \Rightarrow f(s'') \in \mathcal{I}_f(s^\star), \tag{3.15}$$

and that $\mathcal{I}_f(s^\star)$ is optimal.

Case (i) Consider the case when $a \neq \mathsf{sense}_f$. Let us fix a s'' such that $\mathsf{K}(s'', s^\star)$. Note that since K is reflexive we can be sure that such a s'' exists. Therefore, by the successor state axiom for K (equation 3.6) there is an s' such that

$$s'' = \mathsf{do}(a, s') \wedge \mathsf{K}(s', s).$$

By induction we can thus infer that

$$f(s') \in \mathcal{I}_f(s).$$

Now by the successor state axiom for f (equation 3.13) we have that $f(s'') = f(s')$, which gives us that

$$f(s'') \in \mathcal{I}_f(s).$$

Then by the successor state axiom for $\mathcal{I}_f$ (equation 3.14) $\mathcal{I}_f(s^\star) = \mathcal{I}_f(s)$, we have that

$$f(s'') \in \mathcal{I}_f(s^\star),$$

as required for validity. This also shows that to be a valid interval for $\mathcal{I}_f(s^\star)$ the interval must also be a valid interval for $\mathcal{I}_f(s)$. Now by the assumption of optimality any interval narrower than $\mathcal{I}_f(s)$ would no longer be valid. Therefore, $\mathcal{I}_f(s)$ is also the narrowest valid interval for $\mathcal{I}_f(s^\star)$.

Case (ii) Similarly, when $a = \mathsf{sense}_f$ then by the successor state axiom for K (equation 3.6), there is an s' such that

$$s'' = \mathsf{do}(a, s') \wedge \mathsf{K}(s', s) \wedge f(s') = f(s).$$

Therefore,

$$f(s') \in \langle f(s), f(s) \rangle.$$

Now by the successor state axiom for f (equation 3.13) we have that $f(s'') = f(s')$, which gives us that

$$f(s'') \in \langle f(s), f(s) \rangle.$$

Then by the successor state axiom for $\mathcal{I}_f$ (equation 3.14) $\mathcal{I}_f(s^\star) = \langle f(s), f(s) \rangle$, we have that

$$f(s'') \in \mathcal{I}_f(s^\star).$$

as required for validity. To show optimality consider that the width of $\langle f(s), f(s) \rangle$ is 0. Therefore, there can be no narrower interval and so the interval must also be optimal.

□

As a corollary we have that the definition of *Knows* given in equation 3.3 is equivalent to the one given in equation 3.11.

Corollary 3.11.1. *For any sensory fluent f we have that:*

$$\mathit{Knows}(f = c, s) \Leftrightarrow \mathit{Knows}'(f = c, s).$$

Proof. Let us assume $\mathit{Knows}(f = c, s)$. By equation 3.3 this is equivalent to:

$$\forall s' \ \mathsf{K}(s', s) \Rightarrow f(s') = c.$$

Now by theorem 3.11.1, $\mathcal{I}_f$ is valid and optimal, therefore $\mathcal{I}_f(s) = \langle c, c\rangle$, which by equation 3.11 is the definition of $\mathit{Knows}'(f = c, s)$.

Now let us assume $\mathit{Knows}'(f = c, s)$, then by equation 3.11 we have that $\mathcal{I}_f(s) = \langle c, c\rangle$. Once again by applying theorem 3.11.1 we must have that

$$\forall s' \ \mathsf{K}(s', s) \Rightarrow f(s') \in \langle c, c\rangle.$$

Since $\langle c, c\rangle$ has width 0 we can re-write this as:

$$\forall s' \ \mathsf{K}(s', s) \Rightarrow f(s') = c,$$

which by equation 3.3 is the definition of $\mathit{Knows}(f = c, s)$, as required. □

In [98] a number of correctness results are proven for *Knows*. The above equivalence means that under the current set of assumptions the correctness results carry over for *Knows'*.

3.12 Operators for interval arithmetic

Back in section 3.8.3 one of our original motivations for introducing intervals was the promise of being able to conveniently calculate what we know about a term from our knowledge of its subcomponents. For example, suppose in a situation s we know the value of a fluent $f(s)$, what do we know about $(f(s))^2$?

The answer to this question leads us to the large and active research area of interval arithmetic. The fundamental principle used is that interval versions of a given function should be guaranteed to bound all possible values of the non-interval version. For example, let us consider a function $\phi : \mathbb{R} \to \mathbb{R}$. The interval version of this function is $\mathcal{I}_\phi : \mathcal{I}_\mathbb{R} \to \mathcal{I}_\mathbb{R}$. The result of applying $\mathcal{I}_\phi$ to some interval $\boldsymbol{x}$ is another interval $\boldsymbol{y} = \mathcal{I}_\phi(\boldsymbol{x})$. We say that the $\boldsymbol{y}$ is a *valid* interval if for every point $x \in \boldsymbol{x}$, we have that $\phi(x) \in \boldsymbol{y}$. Note also that for any valid interval $\boldsymbol{y}$, if $\boldsymbol{y} \subseteq \boldsymbol{y'}$ then, $\boldsymbol{y'}$ is also a valid interval. If, for every interval $\boldsymbol{x}$, $\mathcal{I}_\phi(\boldsymbol{x})$ gives a valid interval then we say that $\mathcal{I}_\phi$ is a *sound interval version* of ϕ.

As we might expect from our previous discussions defining a sound interval version of any function is trivial. In particular, we just let the interval version return the maximal interval of the relevant number system. For example, the function that, for any argument, returns $\langle -\infty, \infty\rangle$ is a sound interval version of any function $\phi : \mathbb{R} \to \mathbb{R}$.

Hence, we see that once again we also need to be concerned about returning intervals that are as narrow as possible. The *optimal interval version* of a function ϕ is thus defined to be the *sound interval version* that, for every argument, returns the smallest valid interval. Unfortunately, for most interesting functions, no such interval versions are known to exist. There are three basic approaches that have been found to address this shortcoming:

Special Forms Consider the expression $t + (50 - t)$. If we naïvely evaluate this expression for the interval $\langle 0, 50\rangle$ we get back the interval $\langle 0, 100\rangle$. It is clear, however, that the expression simplifies to 50 and the optimal interval is thus $\langle 50, 50\rangle$. Therefore, researchers have looked at various standard forms for

expressions in an attempt to give better results when evaluating the expression using intervals. In general, however, not only is there no known optimal form but there is also no known single form that is always guaranteed to give the best result. The closest researchers have been able to do so far is the so called "centered forms" [1].

Subdivision The standard tool in the interval arithmetic arsenal is subdivision. Suppose we have an interval $\boldsymbol{x}$ and we evaluate $\mathcal{I}_\phi(\boldsymbol{x})$ to give us an interval that is too wide. Then we subdivide $\boldsymbol{x}$ into $\boldsymbol{x}_l$ and $\boldsymbol{x}_r$ such that $\boldsymbol{x} = \boldsymbol{x}_l \cup \boldsymbol{x}_r$. We then evaluate each half separately in the hope that $\mathcal{I}_\phi(\boldsymbol{x}_l) \cup \mathcal{I}_\phi(\boldsymbol{x}_r) \subset \mathcal{I}_\phi(\boldsymbol{x})$. In practice this usually works well although in theory the functions can be noncomputable in which case any hopes of refining our intervals vanish.

Linear intervals The final approach we mention is a new approach that was recently invented by Jeffrey Tupper [115]. The idea is that instead of using constants to bound an interval we use linear functions. Thus for linear expressions, such as $t + (50 - t)$, we can define operators that are guaranteed to return optimal intervals. Of course, we can then recreate similar problems by considering quadratic expressions but Tupper also shows how we can generalize interval arithmetic all the way up to intervals that use general Turing machines as bounds!

3.13 Knowledge of terms

Back in section 3.10 we introduced the abbreviation $\mathcal{I}_{\mathit{Knows}}$. In equation 3.10 we defined $\mathcal{I}_{\mathit{Knows}}$ for fluents and in what follows we shall show how to define $\mathcal{I}_{\mathit{Knows}}$ for general terms. We begin by stating what it means for our definitions to be valid.

Definition 3.13.1 (Validity for terms). *For every term τ, we say that the corresponding interval value of the term given by $\mathcal{I}_{\mathit{Knows}}(\tau, s)$ is a* valid *interval if τ's value in all of the* K*-related situations is contained within it:*

$$\forall s, s' \ \mathsf{K}(s', s) \Rightarrow \tau[s'] \in \mathcal{I}_{\mathit{Knows}}(\tau, s).$$

Fortunately, the general notion of soundness for interval arithmetic carries over into our notion of validity for a $\mathcal{I}_{\mathit{Knows}}$.

Theorem 3.13.1. *Suppose $\mathcal{I}_\phi$ is a sound interval version of an n-ary function $\phi : \mathbb{X}^n \to \mathbb{X}$. Furthermore, let $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{n-1} \in \mathcal{I}_{\mathbb{X}}$ be, respectively, valid intervals for $\mathcal{I}_{\mathit{Knows}}(\tau_0, s), \ldots, \mathcal{I}_{\mathit{Knows}}(\tau_{n-1}, s)$. Then, $\mathcal{I}_\phi(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{n-1})$ is a valid interval for $\mathcal{I}_{\mathit{Knows}}(\phi(\tau_0, \ldots, \tau_{n-1}), s)$.*

Proof. Suppose the theorem is false. Then $\mathcal{I}_\phi(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{n-1})$ is not a valid interval for $\mathcal{I}_{\mathit{Knows}}(\phi(\tau_0, \ldots, \tau_{n-1}), s)$. That is,

$$\forall s' \ \mathsf{K}(s', s) \Rightarrow \phi(\tau_0[s'], \ldots, \tau_{n-1}[s']) \in \mathcal{I}_\phi(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{n-1}),$$

is false. That is, for some K-related situation s' we have that

$$\phi(\tau_0[s'], \ldots, \tau_{n-1}[s']) \notin \mathcal{I}_\phi(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{n-1}).$$

This, however, violates the assumption that $\mathcal{I}_\phi$ is a sound interval version of ϕ. □

The important consequence of this theorem is that our definition of $\mathcal{I}_{\mathit{Knows}}$ for terms can stand upon the shoulders of previous work in interval arithmetic. That is, we can define $\mathcal{I}_{\mathit{Knows}}$ recursively in terms of sound interval versions of functions. Assuming the same assumptions as in theorem 3.13.1 we have that

$$\mathcal{I}_{\mathit{Knows}}(\phi(\tau_0, \ldots, \tau_{n-1}), s) = \mathcal{I}_\phi(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{n-1}).$$

Note that some of our functions may be written using *infix* notation, in which case we may refer to them as *operators*. The important aspect of this definition is that we do not have to redesign a plethora of operators

for interval arithmetic and prove each of them sound. In the previous section we noted the difficulties associated with defining optimal versions of operators. We also noted that there are a number of ways to deal with the problem. Each of the methods we outlines maintains validity and is thus appropriate for us to use. Which particular method we choose to narrow our intervals can be thought of as an implementation issue for our approach.

3.14 Usefulness

For a long list of useful operators for interval arithmetic the reader could do no worse than to consult [115]. By way of example, however, we shall list some useful operators for $\mathcal{I}_{\mathbb{B}}$. Interval versions of operators, and relations, are given in bold. Elsewhere, we rely on context to imply the intended meaning.

Definition 3.14.1 (Operators for $\mathcal{I}_{\mathbb{B}}$).

$$\begin{aligned}
\tau = \langle u, v\rangle &\Leftrightarrow \neg\tau = \langle \neg v, \neg u\rangle \\
\tau_0 = \langle u_0, v_0\rangle \wedge \tau_1 = \langle u_1, v_1\rangle &\Rightarrow \tau_0 \wedge \tau_1 \subseteq \langle u_0 \wedge u_1, v_0 \wedge v_1\rangle \\
\tau_0 \wedge \tau_1 = \langle u, v\rangle &\Rightarrow \tau_0 \subseteq \langle u, 1\rangle \\
\tau_0 = \langle u_0, v_0\rangle \wedge \tau_1 = \langle u_1, v_1\rangle &\Rightarrow \tau_0 \vee \tau_1 \subseteq \langle u_0 \vee u_1, v_0 \vee v_1\rangle \\
\tau_0 \vee \tau_1 = \langle u, v\rangle &\Rightarrow \tau_0 \subseteq \langle 0, v\rangle \\
[\exists x\ \tau(x)] = \langle u, v\rangle &\Rightarrow \tau(c) \subseteq \langle 0, v\rangle, \quad \textit{for any constant } c \\
\textit{For some constant } c, \quad \tau(c) = \langle u, v\rangle &\Rightarrow [\exists x\ \tau(x)] \subseteq \langle u, 1\rangle \\
[\forall x\ \tau(x)] = \langle u, v\rangle &\Rightarrow \tau(c) \subseteq \langle u, 1\rangle, \quad \textit{for any constant } c \\
\textit{For some constant } c, \quad \tau(c) = \langle u, v\rangle &\Rightarrow [\forall x\ \tau(x)] \subseteq \langle 0, v\rangle
\end{aligned}$$

These definitions enable us to evaluate $\mathcal{I}_{\mathsf{Knows}}$ for terms taking on values in $\mathbb{B}$. Notice however that most of the definitions are in terms of $\subseteq$. This is because we can, in general, only guarantee valid results, not optimal ones. For example, if we assume $\tau = \langle 0, 1\rangle$ then we get that $\tau \vee \neg\tau \subseteq \langle 0, 1\rangle$. While this is valid, it is clearly not optimal. Since there are only two numbers in $\mathbb{B}$ we can subdivide to perform an exhaustive search for the optimal value. That is, let $\tau = \tau_0 \cup \tau_1$, where $\tau_0 = \langle 0, 0\rangle$, and $\tau_1 = \langle 1, 1\rangle$. Now we get that $\tau_0 \vee \neg\tau_0 = \langle 1, 1\rangle$, and $\tau_1 \vee \neg\tau_1 = \langle 1, 1\rangle$. With more variables the exhaustive search approach has worst case exponential complexity. In general it may be observed that if each variable occurs only once in an expression then evaluating it will yield an optimal result. Also if we start with thin intervals then we will also get an optimal result. Finally, for a propositional formula in Blake canonical form [22] evaluation with intervals *always* yields an optimal result [53]. Moreover, all propositional formulas can be converted to this form. Thus we can evaluate propositional formulas in linear time and get optimal results. The catch is that converting propositional formulas to Blake canonical form is NP-hard.

When we consider quantifiers the above rules would not form the basis of a particularly useful procedure for evaluating expressions. We recall that the simplest correct procedure would be the one that always just returns the interval $\langle 0, 1\rangle$. For queries containing quantifiers the procedure that follows from the above rules is almost as useless, except that it works adequately when tell it about a specific instance. It is important to bear in mind however that, in general, we are dealing with problems that are not even computable. Consequently for *any* finite set of rules there will always be problems for which we can do no better than return the maximal interval. One immediate consequence of this is that designers of interval arithmetic packages never need worry about unemployment! Regardless, the rules given above certainly suffice as a simple starting point.

Therefore, as we should expect, intervals do not provide us with a means to magically circumvent complexity problems. What they do provide, however, is the ability to track our progress in solving a problem. For the majority of real world problems, where exact knowledge is not imperative, this will often allow us to stop early once we have a "narrow enough" interval. At the very least we can give up early if convergence is too slow. This should be contrasted to other methods of evaluating expressions where we can never be sure whether the method is completely stuck, or is just about to return the solution.

Let us now consider some more examples in which our interval arithmetic approach can be shown to be useful and valid. We begin with a simple example. Suppose we have two relational fluents P, and Q, and that we know P is true or we know Q is true:

$$\mathit{Knows}(P, s) \vee \mathit{Knows}(Q, s).$$

Using the K-fluent it is not hard to see that this implies that we know P or Q:

$$\mathit{Knows}(P \vee Q, s).$$

Proof. The proof involves expanding out the definition of *Knows*:

$$\mathit{Knows}(P, s) \vee \mathit{Knows}(Q, s) \triangleq (\forall s' \ \mathsf{K}(s', s) \Rightarrow P(s')) \vee (\forall s'' \ \mathsf{K}(s'', s) \Rightarrow P(s'')),$$

and then proceeding by case analysis. First consider the case when:

$$\forall s' \ \mathsf{K}(s', s) \Rightarrow P(s').$$

Then we can weaken the postcondition to give:

$$\forall s' \ \mathsf{K}(s', s) \Rightarrow P(s') \vee Q(s').$$

The other case is symmetrical, and the result follows from the definition of *Knows* given in equation 3.3. □

It is also not hard to see that the implication does *not* hold the other way around. As a counter example, consider the case when we have exactly two K-related situations: s_a and s_b, such that: $P(s_a)$, $\neg Q(s_a)$, $\neg P(s_b)$ and $Q(s_b)$.

Now consider the same example using interval-fluents. Once again we can easily prove that:

$$\mathit{Knows}'(P, s) \vee \mathit{Knows}'(Q, s) \Rightarrow \mathit{Knows}'(P \vee Q, s).$$

Proof. We begin by expanding out definitions:

$$\mathcal{I}_{\mathit{Knows}}(P, s) = \langle 1, 1\rangle \vee \mathcal{I}_{\mathit{Knows}}(Q, s) = \langle 1, 1\rangle,$$

and proceed by case analysis. When:

$$\mathcal{I}_{\mathit{Knows}}(P, s) = \langle 1, 1\rangle,$$

from definitions 3.14.1 we have that:

$$\mathcal{I}_{\mathit{Knows}}(P \vee Q, s) = \langle 1, 1\rangle.$$

The other case is symmetrical, and the result follows from the definition of *Knows*′ given in equation 3.11. □

Conversely, if we start from the assumption:

$$\mathit{Knows}'(P \vee Q, s) \triangleq \mathcal{I}_{\mathit{Knows}}(P \vee Q, s) = \langle 1, 1\rangle.$$

Then, all the definitions 3.14.1 allow us to conclude is tautologies, namely that $\mathcal{I}_{\mathit{Knows}}(P, s) \subseteq \langle 0, 1\rangle$ and $\mathcal{I}_{\mathit{Knows}}(Q, s) \subseteq \langle 0, 1\rangle$. That is we can say nothing about our knowledge of P or our knowledge of Q. So, as we should hope, the implication does *not* hold the other way around.

Let us now consider some more examples. Consider knowing P to be false: $\mathit{Knows}(\neg P, s)$ versus not knowing P: $\neg\mathit{Knows}(P, s)$. Firstly, if we assume that K is reflexive, then we have that:

$$\mathit{Knows}(\neg P, s) \Rightarrow \neg\mathit{Knows}(P, s)$$

Proof. The proof is straightforward: We don't know P if in at least one of the K-related worlds P is false. So, if P is false in all the K-related worlds the result follows. We just have to be careful that there are any K-related worlds at all. This can be inferred from the fact that K is reflexive, so $\mathsf{K}(s, s)$. □

The implication clearly does not hold in the other direction.
Likewise, we have that:

$$\mathit{Knows}'(\neg P, s) \Rightarrow \neg \mathit{Knows}'(P, s)$$

Proof.

$$\begin{array}{rl} & \mathit{Knows}'(\neg P, s) \\ \triangleq & \mathcal{I}_{\mathit{Knows}}(\neg P, s) = \langle 1, 1\rangle \\ \Rightarrow & \mathcal{I}_{\mathit{Knows}}(P, s) = \langle 0, 0\rangle \\ \Rightarrow & \mathcal{I}_{\mathit{Knows}}(P, s) \neq \langle 1, 1\rangle \\ \Rightarrow & \neg\mathcal{I}_{\mathit{Knows}}(P, s) = \langle 1, 1\rangle \end{array}$$

□

And conversely

$$\begin{array}{rl} & \neg\mathit{Knows}'(P, s) \\ \triangleq & \neg\mathcal{I}_{\mathit{Knows}}(P, s) = \langle 1, 1\rangle \\ \Rightarrow & \mathcal{I}_{\mathit{Knows}}(P, s) = \langle 0, 0\rangle \vee \mathcal{I}_{\mathit{Knows}}(P, s) = \langle 0, 1\rangle \end{array}$$

case (i)

$$\begin{array}{rl} & \mathcal{I}_{\mathit{Knows}}(P, s) = \langle 0, 0\rangle \\ \Rightarrow & \mathcal{I}_{\mathit{Knows}}(\neg P, s) = \langle 1, 1\rangle \end{array}$$

but for case (ii)

$$\begin{array}{rl} & \mathcal{I}_{\mathit{Knows}}(P, s) = \langle 0, 1\rangle \\ \Rightarrow & \mathcal{I}_{\mathit{Knows}}(\neg P, s) = \langle 0, 1\rangle \end{array}$$

so as we should hope the implication does not hold the other way around.

Now, consider the example of $\exists x\ \mathit{Knows}(P(x), s)$, versus $\mathit{Knows}(\exists x\ P(x), s)$.
Firstly we have that

$$\exists x\ \mathit{Knows}(P(x), s) \Rightarrow \mathit{Knows}(\exists x\ P(x), s)$$

Proof. $\mathit{Knows}(\exists x\ P(x), s)$ holds if in each K-related situation s' there is a constant $c_{s'}$ such that $P(c_{s'}, s')$ holds. Note, the constant $c_{s'}$ that makes $P(x, s)$ true can be a different constant in each s'. Our assumption, however, is that there is some constant c such that $P(c, s')$ holds in every K-related situation s'. Therefore, in each K-related situation s', we can simply set $c = c_{s'}$, and the result follows. □

The implication clearly does not hold in the other direction.
Now consider the same example using intervals. We also have that:

$$\exists x\ \mathit{Knows}'(P(x), s) \Rightarrow \mathit{Knows}'(\exists x\ P(x), s)$$

Proof.

$$\begin{array}{rl} & \exists x\ \mathit{Knows}'\,(P(x), s) \\ \triangleq & \exists x\ \mathcal{I}_{\mathit{Knows}}\,(P(x), s) = \langle 1, 1\rangle \end{array}$$

Then, for some constant c, we have that

$$\begin{array}{rl} & \mathcal{I}_{\mathit{Knows}}\,(P(c), s) = \langle 1, 1\rangle \\ \Rightarrow & \mathcal{I}_{\mathit{Knows}}\,(\exists x\ P(x), s) \subseteq \langle 1, 1\rangle \\ \Rightarrow & \mathcal{I}_{\mathit{Knows}}\,(\exists x\ P(x), s) = \langle 1, 1\rangle \end{array}$$

□

In the other direction we have that:

$$\begin{array}{rl} & \mathit{Knows}'\,(\exists x\ P(x), s) \\ \triangleq & \mathcal{I}_{\mathit{Knows}}\,(\exists x\ P(x), s) = \langle 1, 1\rangle \\ \Rightarrow & \mathcal{I}_{\mathit{Knows}}\,(P(c), s) \subseteq \langle 0, 1\rangle \end{array}$$

Which is a tautology, from which we can (rightly) conclude nothing.

Finally, in section 3.8.3 we saw that we could make deductions based on *modus ponens*. Fortunately, we can perform similar reasoning with intervals.

Theorem 3.14.1. *Let τ_0 and τ_1 be terms for that take on values in $\mathbb{B}$, such that $\langle u, v\rangle$ is a valid interval value for $\mathcal{I}_{\mathit{Knows}}\,(\tau_0, s)$, and $\tau_0[s] \Rightarrow \tau_1[s]$. Then, $\mathcal{I}_{\mathit{Knows}}\,(\tau_1, s) \subseteq \langle u, 1\rangle$.*

Proof. Since $\langle u, v\rangle$ is a valid value for $\mathcal{I}_{\mathit{Knows}}\,(\tau_0, s)$, by definition 3.13.1, we have that

$$\mathcal{I}_{\mathit{Knows}}\,(\tau_0, s) = \langle u, v\rangle \triangleq \forall s'\ \mathsf{K}(s', s) \Rightarrow \tau_0[s'] \in \langle u, v\rangle.$$

In particular, $u\tau_0[s']$. Also, by the assumption that $\tau_0[s] \Rightarrow \tau_1[s]$ we have that $\tau_0[s'] \leqslant \tau_1[s']$. Hence, $u \leqslant \tau_0[s']\tau_1[s'] \leqslant 1$, to give us that

$$\forall s'\ \mathsf{K}(s', s) \Rightarrow \tau_1[s'] \in \langle u, 1\rangle.$$

Therefore, by definition 3.13.1, $\langle u, 1\rangle$ is a valid interval for $\mathcal{I}_{\mathit{Knows}}\,(\tau_1, s)$, as required. □

3.15 Inaccurate Sensors

In [6], the K-fluent approach is extended to handle noisy sensors. It is worth noting that by redefining *Knows* we can also easily extend our approach to allow for inaccurate sensors. We may say that we know a fluent's value to within some Δ, if the width of the interval is less than twice Δ:

$$\mathit{Knows}\,(\Delta, f = z, s) \triangleq \mathcal{I}_f(s) \subseteq \langle z - \Delta, z + \Delta\rangle. \tag{3.16}$$

If we have a bound of $\pm\Delta$ on the greatest possible error for the sensor that recorded yesterday's temperature then we can state that the value sensed for the temperature is within $\pm\Delta$ of the actual value:

$$\begin{array}{l} \mathit{Poss}\,(a, s) \Rightarrow [\mathcal{I}_{\mathsf{temp}}(\mathit{do}(a, s)) = \langle u, v\rangle \Leftrightarrow \\ \qquad (a = \mathsf{senseTemp} \wedge u = \mathsf{max}(0, \mathsf{temp}(s) - \Delta) \wedge v = \mathsf{temp}(s) + \Delta) \vee \\ \qquad\qquad (a \neq \mathsf{senseTemp} \wedge \mathcal{I}_{\mathsf{temp}}(s) = \langle u, v\rangle)]. \end{array} \tag{3.17}$$

3.16 Sensing Changing Values

Until now, we only considered sensing fluents whose value remains constant. In [98] once a fluent becomes known then it stays known. That is, if the value of a known fluent changes then the character will automatically know the fluents new value. In many cases this is somewhat counterintuitive. For example, if one has checked the temperature once then it is quite natural to assume that after a certain period of time the information may be out of date. That is, we would expect to have to sense the temperature periodically.

Using the epistemic K-fluent to model information becoming out of date corresponds to adding possible worlds back in. Unfortunately, the K-fluent keeps track of a character's knowledge of all the sensory fluents all at once. It can therefore be hard to specify exactly which worlds the character should be adding back into its consideration. In contrast, with intervals there is nothing noteworthy about allowing the particular relevant interval to expand. We must simply ensure that our axioms maintain the state constraint that the interval bounds the actual value of the fluent.

At the extreme we can extend our approach to handle fluents that are constantly changing in unpredictable ways. We can model this with exogenous actions. We assume that the current temperature changes in a completely erratic and unpredictable way, according to some exogenous action setTemp. Then, we can write a successor-state axiom for temp that simply states that the temperature is whatever it was set to:

$$\begin{aligned} \mathit{Poss}(a, s) \Rightarrow \mathsf{temp}(\mathit{do}(a, s)) = z \Leftrightarrow \\ [(a = \mathsf{setTemp}(z)) \vee (a \neq \mathsf{setTemp} \wedge \mathsf{temp}(s) = z)]. \end{aligned}$$

We can, also, write a successor state axiom for $\mathcal{I}_{\mathsf{temp}}$. In particular, if we again assume accurate sensors, we can state that the temperature is known after sensing it, otherwise, it is completely unknown:

$$\begin{aligned} \mathit{Poss}(a, s) \Rightarrow [\mathcal{I}_{\mathsf{temp}}(\mathit{do}(a, s)) = \langle u, v\rangle \Leftrightarrow \\ (a = \mathsf{senseTemp} \wedge u = v = \mathsf{temp}(s)) \vee (a \neq \mathsf{senseTemp} \wedge u = 0 \wedge v = \infty)]. \end{aligned} \quad (3.18)$$

Note that this definition works because, by definition, $\forall s\ \mathsf{temp}(s) \in \langle 0, \infty\rangle$. At first glance it may appear strange that we have, for example, $\mathcal{I}_{\mathsf{temp}}(\mathit{do}(\mathsf{setTemp}(2), s)) = \langle 0, \infty\rangle$. Upon reflection, however, the reader will hopefully recall that our intention is to use the IVE fluents to model a character's knowledge of its world. Therefore, until sensing, the character rightly remains oblivious as to the effect of the exogenous action setTemp. For the fluent that keeps track of the temperature in the virtual world we of course get that $\mathsf{temp}(\mathit{do}(\mathsf{setTemp}(2), s)) = 2$.

If we have a bound on the maximum rate of temperature change, per unit time, to be Δ_{temp}, and we add the ability to track the time to our axiomatization, then we can do a lot better. Suppose we have an action tick that occurs once per unit of time. Moreover, we limit exogenous actions to only occurring directly before a tick action. Then we can have a successor-state axiom that states the temperature is known after sensing; or after a period of time it is known to have changed by less than some maximum amount; otherwise it is unchanged:

$$\begin{aligned} \mathit{Poss}(a, s) \Rightarrow [\mathcal{I}_{\mathsf{temp}}(\mathit{do}(a, s)) = \langle u, v\rangle \Leftrightarrow \\ (a = \mathsf{senseTemp} \wedge u = v = \mathsf{temp}(s)) \vee \\ (a = \mathsf{tick} \wedge \exists u_p, v_p\ \mathcal{I}_{\mathsf{temp}}(s) = \langle u_p, v_p\rangle \wedge \\ u = \mathsf{max}(0, u_p - \Delta_{\mathsf{temp}}) \wedge v = v_p + \Delta_{\mathsf{temp}}) \vee \\ (a \neq \mathsf{senseTemp} \wedge a \neq \mathsf{tick} \wedge \mathcal{I}_{\mathsf{temp}}(s) = \langle u, v\rangle]. \end{aligned} \quad (3.19)$$

Note that, this example relies on the underlying notion of discrete change within the situation calculus.

3.17 Extensions

For the case of knowledge of fluents, we have shown that our approach is comparable to previous approaches. The advantages we are claiming for our approach is that it is simple to understand, easy and natural to extend, and trivial to implement.

Beyond this, there are other areas of possible extensions to the situation calculus that we may benefit from considering. We have achieved all the results in this document without these extensions. Therefore, we do not consider these extensions essential to producing animations. For each of the extensions, we shall explain why.

Continuous Processes In [91], the situation calculus is extended to provide a representation of time and event occurrences. This is done by defining a time line corresponding to a sequence, beginning with s_0, of situations. The sequence of situations, called *actual situations* are totally ordered, and the actions which lead to different actual situations are said to have *occurred*. For physics-based applications, such an extension would allow us to consider giving our characters some knowledge about the underlying physics. However, for applications in which continuous processes might be a useful abstraction we have concentrated on very high-level discrete behavior. That is, it seems reasonable to discretize the world when performing high-level planning tasks. The resulting actions can then embed the smooth actions that are required to make things look more realistic.

It is worth pointing out that we can use complex actions to simulate continuous actions. This finite differential approach is standard practice in writing programs that perform mathematical computation. For example, suppose we have a would-be continuous action continuousMove, then we can define this as a procedure something like the following:

```
proc continuousMove
  while ¬Arrived do
    moveALittleBit ⨾
  od
end
```

Concurrency The notion of concurrent actions are introduced into the situation calculus in [66]. Concurrency is handled by interleaving. For example, we say that the concurrent execution of load and aim occurs between two states if and only if both load and aim occur, interleaved in some fashion, between the two states.

In all our animations containing multiple characters, each character is autonomous. Thus, we have emergent concurrent behavior. The point is that we do not explicitly *specify* the concurrent behavior anywhere. While it would clearly be interesting and useful to add such an ability, emergent behavior is a powerful, albeit clumsy tool. It is widely hailed as a feature in other works in behavior animation, and so we shall, with the above qualification, adopt the same stance.

Mental States Above, we discussed knowledge-producing actions. There remain, however, a wealth of additional difficult concepts to consider, such as goals and intentions [35, 100], hysteresis, memory [69] etc.

Once again, it is not the case that our characters do not have intentions, goals, etc. It is just that they are not explicitly represented by the character in a manner that would allow them to reason about them. Given that previous work in the field has not made any attempt to explicitly represent any aspect of character behavior, we feel that our work is an important step in the right direction. More complicated and open problems in knowledge representation can certainly wait for our attention.

User Interaction It is worth pointing out that an understanding of the underlying mathematics is not required of a user seeking to build a character with our approach. That is, it is possible to add some "syntactic sugar" to make an interaction language that is much closer to English. This will be much

easier to use for nonmathematically minded users but will still have a clear mapping to the situation calculus.

It is a simple matter to change the lexical analyzer and parser to accommodate changes of syntax in the input language. Of course, moving further along this path toward natural language interaction is a far more challenging problem. It is also a problem that is a large and distinct research area. We prefer, therefore, to concentrate on the underlying mechanism a character may use to represent knowledge of its virtual world. We leave user interface design and natural language processing to those better qualified for its pursuit.

Chapter 4

Kinematic Applications

In this chapter we shall first introduce how the situation calculus can be used in computer animation. We choose kinematic animation as our initial application because it provides a cleaner canvas for our exposition. One of our main aims in this chapter is to demonstrate how we can use nondeterminism to succinctly specify controllers. The examples we employ will be of increasing complexity and this will lead us in smoothly to chapter 5 where we will discuss physics-based applications.

4.1 Methodology

It should not be surprising that the situation calculus can be used within the context of kinematic animation. There are no obvious impediments to using the facts that are true of a given situation to directly drive an animation. That is, by introducing fluents for any property that we wish to visualize, it is straightforward to produce an animation. For example, we can introduce fluents for position, and orientation and then define successor state axioms to describe how they change over time.

4.2 Example

Our first example is of some brightly colored airplanes flying in an interesting pattern. The first step in our formalization is to define some fluents, the intended use of which is alluded to by their names.

$\mathsf{speed}(s)$ – the plane's angular velocity around a unit circle
$\mathsf{theta}(s)$ – the plane's angular displacement about a unit circle
$\mathsf{pos}_x(s)$ – the plane's position
$\mathsf{pos}_y(s)$
$\mathsf{pos}_z(s)$
$\mathsf{angle}(s)$ – the plane's angle of rotation about the z-axis

The only action in our ontology is:

update – updates all the fluents

The pre-condition axioms state that the actions are always possible, so we shall omit them. The successor state axioms state that after each update action the angle is incremented by the current angular velocity, and that the angular velocity stays constant.

$$\mathit{Poss}(a, s) \Rightarrow [\mathsf{theta}(\mathit{do}(a, s)) = \theta_{\text{new}} \Leftrightarrow \exists \theta_{\text{old}}, \theta_{\text{new}}, \dot{\theta}\ \mathsf{theta}(s) = \theta_{\text{old}} \wedge \mathsf{speed}(s) = \dot{\theta} \wedge$$
$$((a = \mathsf{update} \wedge \theta_{\text{new}} = \theta_{\text{old}} + \dot{\theta}) \vee (\theta_{\text{new}} = \theta_{\text{old}}))];$$
$$\mathit{Poss}(a, s) \Rightarrow [\mathsf{speed}(\mathit{do}(a, s)) = \mathsf{speed}(s)]$$

The initial state is given as:

$$\exists x, y\ \mathsf{speed}(\mathrm{s}_0) = y \wedge \mathrm{random}() = x \wedge$$
$$((\mathrm{Even}(x) \wedge y = \frac{k_0 + x \bmod k_0}{k_1}) \vee (\mathrm{Odd}(x) \wedge y = -\frac{k_0 + x \bmod k_0}{k_1}))$$
$$\mathsf{theta}(\mathrm{s}_0) = k_2 \mathrm{random}() \bmod k_3$$

In this simple example the remaining fluents are just *defined* fluents. That is, they are defined entirely in terms of the previously mentioned fluents and are not mentioned in other definitions. Thus in any formula they can always be replaced, without fear of causing problems, by their definition. The definitions are designed to result in an interesting flight pattern when $(\mathsf{pos}_x(s), \mathsf{pos}_y(s), \mathsf{pos}_z(s))$ is used for the plane's position and $\mathsf{angle}(s)$ gives the plane's rotation around the z-axis.

$$\begin{array}{lcl}
\mathsf{pos}_x(s) & \triangleq & k_4 \sin(k_5 \mathsf{theta}(s)) \\
\mathsf{pos}_y(s) & \triangleq & k_6 \sin(k_7 \mathsf{theta}(s)) \\
\mathsf{pos}_z(s) & \triangleq & k_8 + k_9 \cos(\mathsf{theta}(s)) \\
\mathsf{angle}(s) & \triangleq & k_{10} \frac{|\mathsf{speed}(s)| + \mathsf{speed}(s)}{k_{11} |\mathsf{speed}(s)|} + k_{12}(k_{13} \sin(\mathsf{theta}) - k_{14})
\end{array}$$

The following complex action can then be used to generate the required animation:

while $(true)$ **do**
  $\mathsf{update}\ ^{\circ}_{9}$
end

We can easily extend the example to multiple planes by indexing all the fluents. So, for example, $\mathsf{theta}(i, s)$ is the angular displacement of plane i. Our formalization was based on the existing code written by Kilgard as a demonstration of OpenGL® [125]. Figure 4.1 shows some frames from an animation. The values for the constants k_i used to generate the animation were as follows:

$$\begin{array}{lll}
k_0 = 20, & k_1 = 1000, & k_2 = \frac{1111}{10000}, \\
k_3 = 257, & k_4 = 4, & k_5 = 2, \\
k_6 = 3, & k_7 = \frac{17}{5}, & k_8 = -9, \\
k_9 = 4, & k_{10} = 180, & k_{11} = 2, \\
k_{12} = \frac{180}{\pi}, & k_{13} = (\arctan 2 + \frac{\pi}{2}), & k_{14} = \frac{\pi}{2}.
\end{array}$$

4.3 Utilizing Non-determinism

In the above example every detail of the character's behavior is completely specified. One of the key advantages to our approach, however, is the ability to omit details from the behavioral specifications and have the computer fill in the details. That is, the logical basis of our specifications allows us to give our characters reasoning abilities, to automatically fill in some details that we may choose to omit. This makes it straightforward to build, reconfigure or extend the behavior control system of the creatures. It is ideal for applications where animations need only be generated once using powerful computers. When, as with computer games, we wish to build fast, reusable behavioral controllers our approach lends itself to a incremental style of development. The developer can quickly, and easily generate prototypes using high-level specifications. The creatures can use their background knowledge and reasoning abilities to automatically fill in the low-level details. Then, if required, the specifications can be refined to be more efficient. This is done by adding, in a form that is chosen to be similar to conventional programming constructs, control information to direct the character's search for appropriate behavior. The control information is like a "sketch plan" of how the agent should behave. We shall often refer to the sketch plans as "advice", by which we simply mean that our instructions constrain the search area in which the character looks for suitable behavior.

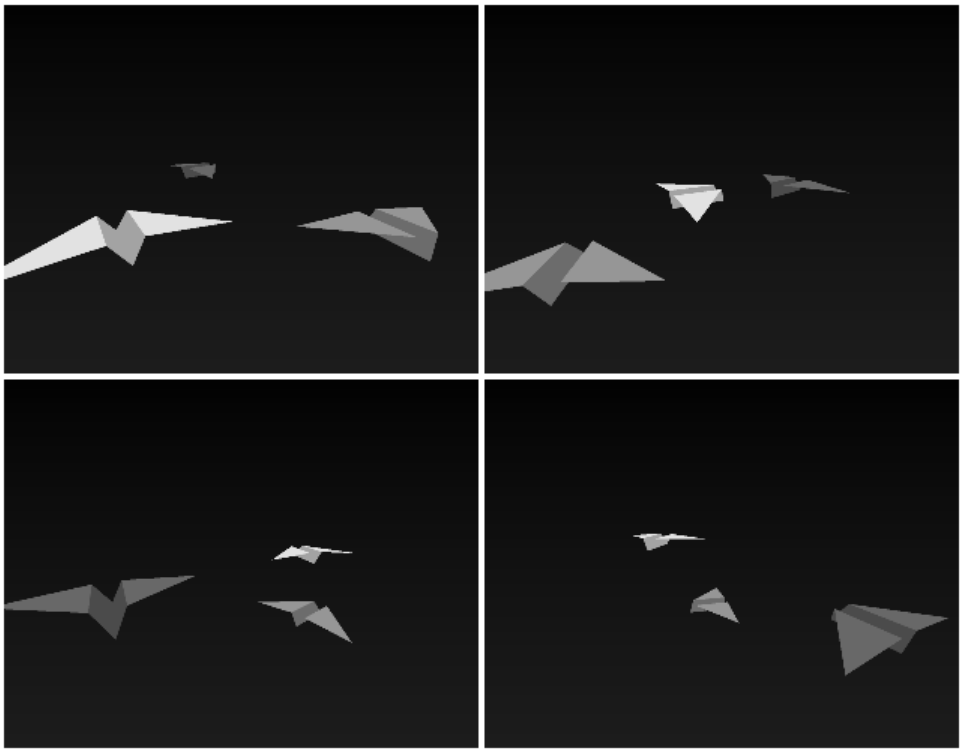


Figure 4.1: Some frames from a simple airplane animation.

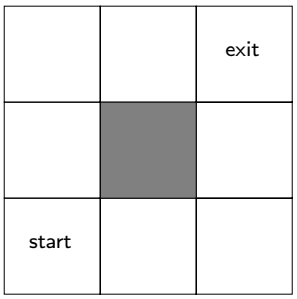
4.4 Another example

For our second example we shall retain the previous geometric model but try to tackle a more complicated task than flying around in a pretty pattern. We shall choose a classic maze solving problem as our vehicle of erudition.

Let us suppose we have a maze defined by a predicate $\mathsf{Free}(c)$, that holds when and only when the grid cell c is "free". That is it is within range and is not occupied by an obstacle:

$$\mathsf{Free}(c) \Leftrightarrow \mathsf{InRange}(c) \wedge \neg\mathsf{Occupied}(c), \quad \text{where}$$
$$\mathsf{InRange}((x,y)) \Leftrightarrow 0 \leqslant x < \mathsf{size}_x \wedge 0 \leqslant y < \mathsf{size}_y$$

$\mathsf{Occupied}(c)$, size_x, and size_y each depend upon the maze in question. In addition, there two maze dependent constants start and exit that specify the entry and exit points of a maze. Figure 4.2 shows a simple maze and the corresponding definition.



$\mathsf{start} = (0,0)$
$\mathsf{exit} = (2,2)$
$\mathsf{size}_x = 3$
$\mathsf{size}_y = 3$
$\mathsf{Occupied} = (1,1)$

Figure 4.2: A simple maze.

We also need to define some functions that describe a path within the maze. We say that the adjacent cell "north" of a given cell is the one directly above it, similarly for "south", "east" and "west".

$$\mathsf{adjacent}((x,y),d) = \begin{cases} (x+1,y) & \text{if } d = \text{north} \\ (x-1,y) & \text{if } d = \text{south} \\ (x,y+1) & \text{if } d = \text{east} \\ (x,y-1) & \text{if } d = \text{west} \end{cases}$$

This completes the axiomatization of the background domain.

We now introduce a fluent position that gives the position in the current situation. The list of cells visited so far is given by the defined fluent $\mathsf{visited}(s)$. It is defined recursively on the situation to be the list of all the positions in previous situations.[1]

$$\mathsf{visited}(\mathrm{s}_0) \triangleq []$$

$$\mathsf{visited}(\mathit{do}(a,s)) \triangleq \begin{cases} [\mathsf{position}(s)|\mathsf{visited}(s)] & \text{if } \exists d\ a = \mathsf{move}(d) \\ \mathsf{visited}(s) & \text{otherwise} \end{cases}$$

For example, in figure 4.3, when $s = \mathit{do}(\mathsf{move}(\mathsf{east}), \mathit{do}(\mathsf{move}(\mathsf{east}), \mathit{do}(\mathsf{move}(\mathsf{north}), \mathrm{s}_0)))$, we have that $\mathsf{position}(s) = (2,1)$, and that $\mathsf{visited}(s) = [(2,0),(1,0),(0,0)]$.

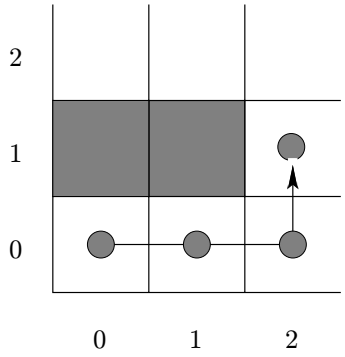


Figure 4.3: Visited cells.

Next we introduce a move action $\mathsf{move}(d)$, that corresponds to moving in direction d. It is possible to move to a new cell provided it is free and has not been visited before.

$$\mathit{Poss}(\mathsf{move}(d,s)) \Leftrightarrow \exists c\ c = \mathsf{adjacent}(\mathsf{position}(s),d) \wedge \mathsf{Free}(c) \wedge c \notin \mathsf{visited}(s)$$

So for example, in figure 4.5, if we have previously been to the location marked with the filled dot, and in situation s the character is at the location marked with the unfilled dot, then it is only possible to move north, south or east:

$$\exists d\ \mathit{Poss}(\mathsf{move}(d),s) \Leftrightarrow d = \mathsf{north} \vee d = \mathsf{south} \vee d = \mathsf{east}$$

It is important to note that, in general, it is possible to move to more than one cell. This fact will be exploited later on in the specification of a path through the maze. In contrast, as in figure 4.4, it may be the case that is only possible to move in one particular direction.

The successor state axiom for position is given below. It states that, provided the action is possible, the position after performing a move action is the cell adjacent to the previous position, in the direction of the move.

$$\mathit{Poss}(a,s) \Rightarrow [\mathsf{position}(\mathit{do}(a,s)) = p' \Leftrightarrow (\exists d\ a = \mathsf{move}(d) \wedge p' = \mathsf{adjacent}(\mathsf{position}(s),d)) \vee (\neg\exists d\ a = \mathsf{move}(d) \wedge p' = \mathsf{position}(s))]$$

[1] We use standard Prolog notation for lists.

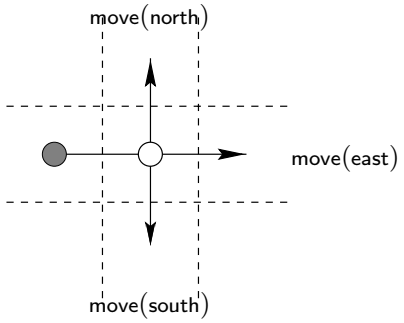


Figure 4.4: Choice of possibilities for a next cell to move to.

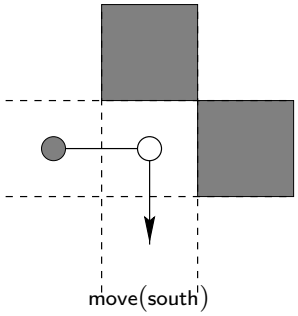


Figure 4.5: Just one possibility for a next cell to move to.

So for example, in figure 4.6, if we have previously been to the locations marked with the filled dots, and in situation s the character moves north to the unfilled dot, then we have that position$(s) = (2, 0)$ and that position$(do(\text{move}(\text{north}), s)) = (2, 1)$.

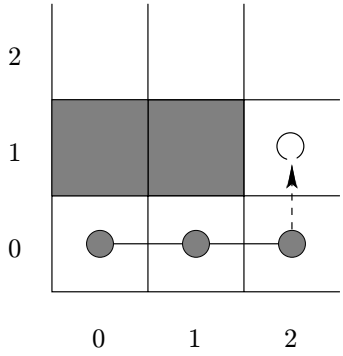


Figure 4.6: Updating maze fluents.

The initial state is given in terms of the maze definition.

$$\text{position}(\text{s}_0) = \text{start}$$

This completes the definition of the primitive actions. Next we define the complex actions that allow us to specify a path through the maze. The idea is that a path consists of a sequence of choices about which way to turn that result in the position becoming equal to the exit point of the maze.

while position $\neq$ exit **do**
  $(\pi\ d)$ move(d)
od

Note that the use of regular programming constructs may initially cause confusion to the reader of the above code. It is important to bear in mind that the $(\pi\ d)$ construct should be read as "pick the *correct* direction d". Perusing the definitions given in appendix D may serve to alleviate any sense of bewilderment. To make things even clearer we shall, however, consider the expansion of the complex actions in terms of their definitions. We shall consider the simple maze described previously in figure 4.2.

In the initial situation we have that:

$$\mathsf{position}(\mathrm{s}_0) \neq \mathsf{exit}$$

Thus the guard of the "while" loop holds and we can try to expand $\mathit{Do}((\pi\ d)\ \mathsf{move}(d), \mathrm{s}_0, s)$. Expanding this out into the full definition gives:

$$\begin{aligned}
&(\mathit{Poss}(\mathsf{move}(\mathsf{north}), \mathrm{s}_0) \wedge s = \mathit{do}(\mathsf{move}(\mathsf{north}), \mathrm{s}_0)) \vee \\
&(\mathit{Poss}(\mathsf{move}(\mathsf{south}), \mathrm{s}_0) \wedge s = \mathit{do}(\mathsf{move}(\mathsf{south}), \mathrm{s}_0)) \vee \\
&(\mathit{Poss}(\mathsf{move}(\mathsf{east}), \mathrm{s}_0) \wedge s = \mathit{do}(\mathsf{move}(\mathsf{east}), \mathrm{s}_0)) \vee \\
&(\mathit{Poss}(\mathsf{move}(\mathsf{west}), \mathrm{s}_0) \wedge s = \mathit{do}(\mathsf{move}(\mathsf{west}), \mathrm{s}_0))
\end{aligned}$$

However, from the action preconditions for Poss and the definition of the maze we can see that:

$$\mathit{Poss}(\mathsf{move}(\mathsf{north})) \wedge \neg\mathit{Poss}(\mathsf{move}(\mathsf{south})) \wedge \mathit{Poss}(\mathsf{move}(\mathsf{east})) \wedge \neg\mathit{Poss}(\mathsf{move}(\mathsf{west}))$$

This leaves us with

$$s = \mathit{do}(\mathsf{move}(\mathsf{north}), \mathrm{s}_0) \vee s = \mathit{do}(\mathsf{move}(\mathsf{east}), \mathrm{s}_0).$$

That is there are two possible resulting situations. That is why we refer to this style of program as nondeterministic.

In contrast, in situation $s = \mathit{do}(\mathsf{move}(\mathsf{north}), \mathrm{s}_0)$ there is only one possible resulting situation. We have that $\mathit{Do}((\pi\ d)\ \mathsf{move}(d), s, s')$ expands out into $s' = \mathit{do}(\mathsf{move}(\mathsf{north}), s)$.

If we expand out the macro $\mathit{Do}(\mathbf{while}\ \mathsf{position} \neq \mathsf{exit}\ \mathbf{do}(\pi\ d)\ \mathsf{move}(d)\ \mathbf{od}, \mathrm{s}_0, s_f)$ from start to finish we get that:

$$\begin{aligned}
s_f \quad = \quad & \mathit{do}(\mathsf{move}(\mathsf{east}), \\
& \quad \mathit{do}(\mathsf{move}(\mathsf{east}), \\
& \quad\quad \mathit{do}(\mathsf{move}(\mathsf{north}), \\
& \quad\quad\quad \mathit{do}(\mathsf{move}(\mathsf{north}), \mathrm{s}_0)))) \vee \\
s_f \quad = \quad & \mathit{do}(\mathsf{move}(\mathsf{north}), \\
& \quad \mathit{do}(\mathsf{move}(\mathsf{north}), \\
& \quad\quad \mathit{do}(\mathsf{move}(\mathsf{east}), \\
& \quad\quad\quad \mathit{do}(\mathsf{move}(\mathsf{east}), \mathrm{s}_0)))).
\end{aligned}$$

So our "program" does indeed specify all paths through the maze.

4.4.1 Implementation

By running the above program through our compiler, and rewriting the precondition and effect axioms in Prolog we can have the computer automatically search through the set of possible situations to find paths through a maze. Using depth-first search on the length possible future situations we can generate a path through the maze shown in figure 4.7 in a few seconds.

It is worth noting that the "program" given by the complex action 4.1 is a specification of a path through a maze. The *specification* of a path does not include backtracking. However, this does not mean that when *searching* for a path we cannot backtrack. In fact this is exactly how the Prolog interpreter searches for a

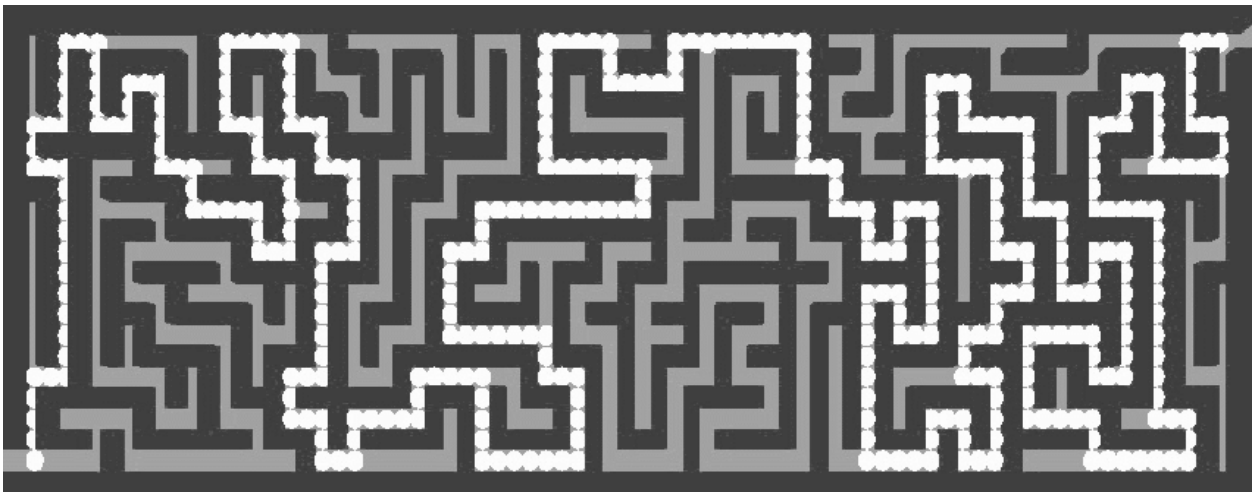


Figure 4.7: A path through a maze.

path. It tries to go forward for as long as it can, when it gets stuck it backtracks to the last point it had a choice and tries the other route. Of course, if the interpreter does eventually find a path, then by definition, it will not contain cycles.

Although our maze solving program is very succinct it is not necessarily very efficient. We can easily start to reduce some of the nondeterminism by specifying a "best-first" search strategy. In this approach we will not leave it up to the computer to decide how to search the possible paths but constrain it to investigate paths first that head toward the exit. This requires extra lines of code but will result in faster execution.

For example, suppose we add an action $\mathsf{goodMove}(d)$, such that it is possible to move to cell d if it is possible to "move" there and d is closer to the goal than we are now:

$$\mathit{Poss}\,(\mathsf{goodMove}(d), s) \Leftrightarrow \mathit{Poss}\,(\mathsf{move}(d), s) \wedge \mathsf{Closer}(\mathsf{exit}, d, \mathsf{position}(s)).$$

Now we can rewrite our high-level controller as one that prefers to move toward the exit position whenever possible:

while position $\neq$ exit **do**
  if $\exists d$ *Poss* (goodMove(d)) **then**
    (π d) goodMove(d)
  else
    (π d) move(d)
  fi
od

At the extreme there is nothing to prevent us from coding in a simple deterministic strategy such as the "left-hand" rule. See [84] for some further alternatives for maze solving. The important point is not that our approach rules out any of the algorithms one might consider when writing the same program in C. Rather it opens up new possibilities for very high-level specifications of behavior. Admittedly these might not be very efficient, but for rapid prototyping, or for one-off animations, they may suffice. Moreover, if they do suffice then we may have saved ourselves a great deal of programming effort.

4.4.2 Intelligent Flocks

The simplest way to add multiple characters to the same scene is to use defined fluents. That is we treat a group of characters as a single entity. For example, we might specify that the center of the group should be at a specified location. We can achieve such an effect by defining axioms that correspond to the action of moving the group center. The effect of the primitive actions on the individual group members is given by defined fluents in terms of the center of the group. Thus the group as a whole can behave in specified ways and perform complicated reasoning tasks.

To control multiple characters in the scene as cooperating individuals is much more complex. In chapter 5 we will see examples of multiple autonomous agents with in the scene. The behavior that results from the interaction between them is *emergent*. That is to say it arises from the interaction of their respective behavior rules, and the environment. There is no attempt to explicitly state the behavior we expect from the group as a whole when the individuals are behaving autonomously. In the interim one might imagine a behavior culling scheme in which characters cease to be treated as individuals once they achieve a certain proximity. This could be achieved quite simply with a fluent InGroup that was true when the character was part of a group. The characters behavior could then be predicated on whether or not it was part of a group.

It seems unlikely to us, however, that scenes with over twenty characters would often require highly sophisticated behavior from each individual character. We could imagine using some level of detail measure to cull unnecessary character behaviors. For example, for a few foreground characters we could use our logical reasoning approach to control their behavior. The remaining "background" characters could utilize a more reactive approach. Other criteria for culling behaviors could also include notions of crowd membership. Individual characters within a crowd would become more reactive and we could treat the whole crowd or "flock" as a single intelligent entity.

4.5 Camera Control

We now turn our attention to a real-world kinematic motion specification problem. In particular we focus on the problem of camera placement within a scene. The inspiration for this application comes from two recent papers on the subject [56, 30]. These two papers use a simple scripting language to implement hierarchical finite state machines for camera control. Their work was, in turn, inspired by standard texts on cinematography, notably [4].

To understand what follows the reader will require some rudimentary knowledge of cinematography. The exposition given in [56] will suffice for our purposes:

> Although a film can be considered to be nothing more than a linear sequence of frames, it is helpful to consider it as having a structure. At the highest level, a film is a sequence of scenes, each of which captures a specific situation or action. Each scene, in turn, is composed of one or more shots. A single shot covers the small portion of a movie between when a camera is turned on and when it is turned off. Typically, a film is comprised of a large number of individual shots, with each shot lasting from a second or two to tens of seconds.
>
> Directors specify camera placements relative to the Line, an imaginary vector connecting two interacting actors, or directed along the line of an actor's motion, or oriented in the direction the actor is facing (Figure 4.8). Shooting actor X from camera position b is called a parallel camera placement. Shooting X from position c yields an internal reverse placement; typically, in this setup, X occupies only the left two-thirds of the screen. Shooting from position d results in an apex shot that shows both actors. Finally, shooting from g is called an external reverse placement; the left two-thirds of the screen shows actor X in focus while the right one-third shows the back side of actor Y's head.

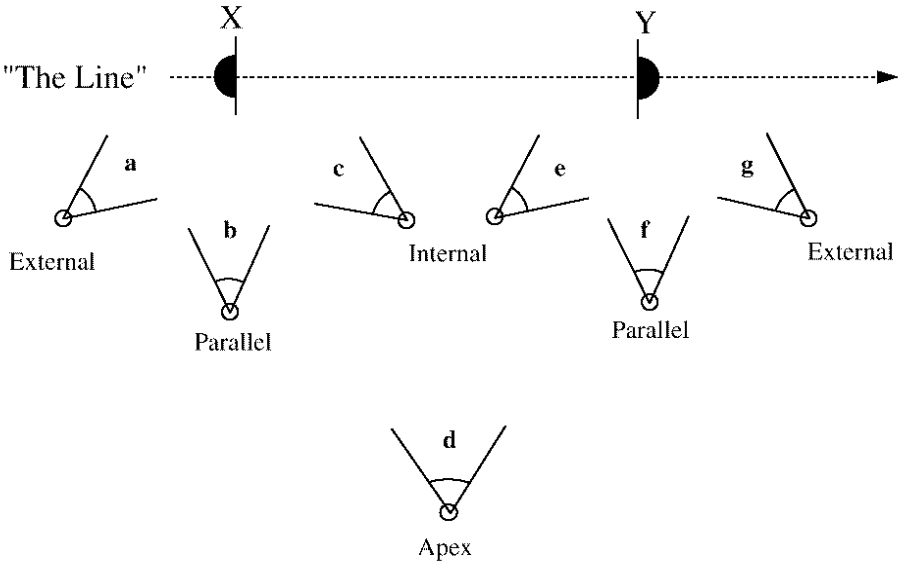


Figure 4.8: Camera placement is specified relative to "the Line" (Adapted from figure 1 of [He96]).

4.5.1 Axioms

In what follows we assume that the motion of all other objects in the scene has been computed. Our task is to decide, for each frame, the vantage point from which it is to be rendered.

The precomputed scene is formalized as a lookup function $\mathsf{configuration} : \textsc{object} \times \mathbb{N} \to \mathbb{R}^n$, which for each object obj, and each frame $\mathsf{frameNo}$ returns a list $\mathsf{configuration}(\mathsf{obj}, \mathsf{frameNo})$ of the n numbers that can completely specify the position, orientation, and shape of the object obj at frame $\mathsf{frameNo}$.

We have a fluent $\mathsf{frame} : \textsc{situation} \to \mathbb{N}$ that keeps track of the current frame number. Initially the frame number is zero: $\mathsf{frame}(S_0) = 0$. The successor state axiom for frame just states that a tick action is always possible and its effect is to cause the frame number to be incremented by one:

$$\mathsf{frame}(do(a, s)) = k \Leftrightarrow (a = \mathsf{tick} \wedge k = \mathsf{frame}(s) + 1) \vee k = \mathsf{frame}(s).$$

The defined fluent $\mathsf{scene} : \textsc{object} \times \textsc{situation} \to \mathbb{R}^n$ specifies the n numbers that can completely specify the position, orientation, and shape of the object in the current situation:

$$\mathsf{scene}(\mathsf{obj}, s) \triangleq \mathsf{configuration}(\mathsf{obj}, \mathsf{frame}(s)).$$

The most common camera placements used in cinematography will be modeled in our formalization as primitive actions. In [56] these actions are referred to as "camera modules".

In what follows we adopt the standard OpenGL® conventions for camera specification [125]. That is, we specify the camera with two fluents: $\mathsf{lookFrom} : \textsc{situation} \to \mathbb{R}^3$, and $\mathsf{lookAt} : \textsc{situation} \to \mathbb{R}^3$. Technically we must also specify an up vector. For most applications, however, we will want up to be constant, pointing along the y-axis. Therefore, we shall assume that $\mathsf{up} = (0, 1, 0)$. In addition, we also need to specify the viewing frustum. The standard OpenGL® way of doing this is with a field of view angle, an aspect ratio, a near clipping plane and a far clipping plane. Once again we make the simplifying assumption that they remain fixed.

Despite our simplifications we still have a great deal of flexibility in our specifications. We will now give examples of effect axioms for some of the primitive actions in our ontology. This will result in a useful application. Moreover it should be clear how we could, if we so desired, extend the approach to formalize other aspects of our domain.

The simplest camera placement action is the null action which leaves the camera in its previous configuration. Since it has no effect at all there is nothing to write for the effect axiom.

The raised action provides a placement relative to the previous camera position, whatever it happened to be. The new camera is further back and higher than the old camera, but has the same orientation as the old camera:

$$\mathsf{lookFrom}(s) = \boldsymbol{e} \wedge \mathsf{lookAt}(s) = \boldsymbol{c} \Rightarrow \mathsf{lookFrom}(\mathit{do}(\mathsf{raised}, s)) = \boldsymbol{e} + k_1(\widehat{\boldsymbol{e} - \boldsymbol{c}}),$$

where k_1 is some constant.

The fixed(eye, center) action is used to explicitly specify a particular camera configuration. We can, for example, use it to provide an overview shot of the scene:

$$\mathsf{lookFrom}(\mathit{do}(\mathsf{fixed}(\mathsf{e}, \mathsf{c}), s)) = \boldsymbol{e};$$
$$\mathsf{lookAt}(\mathit{do}(\mathsf{fixed}(\mathsf{e}, \mathsf{c}), s)) = \boldsymbol{c}.$$

A more complicated action is external(A, B). It places the camera so that character A is seen over the shoulder of character B. One effect of this action, therefore, is that the camera is looking at the character A:

$$\mathsf{scene}(A(\mathsf{upperbody}, \mathsf{centroid}), s) = \boldsymbol{p} \Rightarrow \mathsf{lookAt}(\mathit{do}(\mathsf{external}(A, B), s)) = \boldsymbol{p}.$$

The other effect is that the camera is located above character B's shoulder. This might be accomplished with an effect axiom such as:

$$\mathsf{scene}(B(\mathsf{shoulder}, \mathsf{centroid}), s) = \boldsymbol{p} \wedge \mathsf{scene}(A(\mathsf{upperbody}, \mathsf{centroid}), s) = \boldsymbol{c} \Rightarrow$$
$$\mathsf{lookFrom}(\mathit{do}(\mathsf{external}(A, B), s)) = \boldsymbol{p} + k_2\mathsf{up} + k_3(\widehat{\boldsymbol{p} - \boldsymbol{c}})$$

where k_2 and k_3 are some suitable constants.

There are many other possible camera placement actions. Some of them are listed in [56], others may be found in [4]. Many involve performing complicated geometric calculations to obtain the correct position, orientation and field of view of the camera in order to satisfy constraints on the placement and screen coverage of the protagonists in the final image. Such constraints include cinematographic rules relating to "cutting heights" [56] and occlusion problems. To satisfy them it is possible that we may even have to resort to moving the original scene around!

4.5.2 Complex actions

Before we proceed we must introduce some additional concepts from cinematography. Once again we quote from [56]:

> Perhaps the most significant discovery of cinematographers is that there are stereotypical formulas for capturing specific actions as sequences of shots. For example, in a dialogue among three actors, a film maker might begin with an establishing shot of all three people, before moving to an external reverse shots of two of the actors as they converse, interspersing occasional reaction shots of the third actor. Film books (Arijon [4]) provide an informal compilation of formulas, along with a discussion of the situations when a film maker might prefer one formula over another.

In [56] the authors discuss one particular formula for filming two characters talking to one another. The idea is to flip between "external" shots of each character, focusing on the character doing the talking. The shots are interspersed with reaction shots of the other character to break up the monotony of lengthy shots of the speaker. In the paper the formula is encoded as a finite state machine. We will show how elegantly

we can capture the formula using complex actions (see section 3.6). Firstly we must introduce some new fluents.

The fluent $\mathsf{Talking}(A, B)$ is true if a character A is talking to another character B. The effect axioms for Talking simply state that the fluent becomes true when character A starts talking to character B, and it becomes false when they stop:

$$\mathsf{Talking}(A, B, \mathit{do}(\mathsf{startTalk}(A, B)), s),$$
$$\neg\mathsf{Talking}(A, B, \mathit{do}(\mathsf{stopTalk}(A, B)), s).$$

We shall treat startTalk, and stopTalk as exogenous actions. In terms of an implementation these exogenous actions can easily be generated automatically. That is, since our characters are situated in a virtual world any talking must have originally been instigated by the application that we used to generate the configuration function. We can simply modify the macro expansion of the complex actions to look up whether A or B are talking at the frame number given by $\mathsf{frame}(s)$. If either of them are talking then the corresponding talk action is generated. An example of how the macro expansion can be modified is given in section 5.4.2.

Next we introduce a fluent silenceCount to keep count of how long it has been since a character spoke with the following effect axioms:

$$\mathsf{silenceCount}(\mathit{do}(\mathsf{stopTalk}(A, B), s)) = k_a,$$
$$\mathsf{silenceCount}(\mathit{do}(\mathsf{setCount}, s)) = k_a,$$
$$\neg\exists A, B\ \mathsf{Talking}(A, B, s) \Rightarrow \mathsf{silenceCount}(\mathit{do}(\mathsf{tick}, s)) = \mathsf{silenceCount}(s) - 1.$$

Note that k_a is a constant ($k_a = 10$ in [56]), such that after k_a ticks of no-one speaking the counter will be negative. A similar fluent filmCount keeps track of how long the camera has been pointing at the same character:

$$\mathsf{Talking}(A, B, s) \Rightarrow \mathsf{filmCount}(\mathit{do}(\mathsf{setCount}, s)) = k_b,$$
$$\neg\mathsf{Talking}(A, B, s) \Rightarrow \mathsf{filmCount}(\mathit{do}(\mathsf{setCount}, s)) = k_c,$$
$$\mathsf{Talking}(A, B, s) \Rightarrow \mathsf{filmCount}(\mathit{do}(\mathsf{external}(A, B), s)) = k_b,$$
$$\neg\mathsf{Talking}(A, B, s) \Rightarrow \mathsf{filmCount}(\mathit{do}(\mathsf{external}(A, B), s)) = k_c,$$
$$\mathsf{filmCount}(\mathit{do}(\mathsf{tick}, s)) = \mathsf{filmCount}(s) - 1.$$

k_b and k_c are constants ($k_b = 30$ and $k_c = 15$ in [56]) that state how long we can stay with the same shot before the counter becomes negative. Note that the constant for the case of looking at a non-speaking character is lower. We will keep track of which constant we are using with the fluent tooLong:

$$\mathsf{Talking}(A, B, s) \Rightarrow \mathsf{tooLong}(\mathit{do}(\mathsf{external}(A, B), s)) = k_b,$$
$$\neg\mathsf{Talking}(A, B, s) \Rightarrow \mathsf{tooLong}(\mathit{do}(\mathsf{external}(A, B), s)) = k_c.$$

For convenience and ease of understanding we now define two new fluents in terms of our counter fluents:

$$\mathsf{Boring}(s) \triangleq \mathsf{filmCount}(s) < 0,$$
$$\mathsf{TooFast}(s) \triangleq \mathsf{tooLong}(s) - k_s \leqslant \mathsf{filmCount}(s).$$

They capture the notion of when a shot has become boring because it has gone on too long, and when a shot has not gone on long enough. We need the notion of a minimum time for each shot to avoid instability that would result in flitting between one shot and another too quickly.

Finally, we introduce a fluent Filming to keep track of who the camera is pointing at, we have that:

$$\mathsf{Filming}(A, \mathit{do}(\mathsf{external}(A, B), s)),$$
$$\neg\mathsf{Filming}(A, \mathit{do}(\mathsf{external}(B, A), s)).$$

Until now we have not mentioned any preconditions for our actions. The reader may assume that, unless stated otherwise, all actions are always possible. In contrast, the precondition axiom for the external camera

action states that we only want to be able to point the camera at character A if we are already filming A, and it has not got boring yet; or we not filming A, and A is talking, and we have stayed with the current shot long enough:

$$\begin{aligned} \mathit{Poss}\,(\mathsf{external}(A,B),s) \quad \Leftrightarrow \quad & (\neg\mathsf{Boring}(s) \wedge \mathsf{Filming}(A,s)) \vee \\ & (\mathsf{Talking}(A,B,s) \wedge \neg\mathsf{Filming}(A,s) \wedge \neg\mathsf{TooFast}(s)). \end{aligned}$$

We are now in a position to define the controller that will move the camera to look at the character doing the talking, with occasional respites to focus on the other character's reactions:

```
setCount ⨟
while 0 < silenceCount do
  processTalk ⨟
  (π A, B) external(A, B) ⨟
  tick
od
```

As in the path through a maze program 4.1, this specification makes heavy use of the ability to nondeterministically choose arguments. The reader might like to contrast this definition with the encoding given

in [56] to achieve the same result:

```
DEFINE_IDIOM_IN_ACTION(2Talk)
     WHEN ( talking(A, B) )
          DO ( GOTO (1); )
     WHEN ( talking(B, A) )
          DO ( GOTO (2); )
END_IDIOM_IN_ACTION

DEFINE_STATE_ACTIONS(COMMON)
     WHEN (T < 10)
          DO ( STAY; )
     WHEN (!talking(A, B) && !talking(B, A))
          DO ( RETURN; )
END_STATE_ACTIONS

DEFINE_STATE_ACTIONS(1)
     WHEN ( talking(B, A) )
          DO ( GOTO (2); )
     WHEN ( T > 30 )
          DO ( GOTO (4); )
END_STATE_ACTIONS

DEFINE_STATE_ACTIONS(2)
     WHEN ( talking(A, B) )
          DO ( GOTO (1); )
     WHEN ( T > 30 )
          DO ( GOTO (3); )
END_STATE_ACTIONS

DEFINE_STATE_ACTIONS(3)
     WHEN ( talking(A, B) )
          DO ( GOTO (1); )
     WHEN ( talking(B, A) && T > 15 )
          DO ( GOTO (2); )
END_STATE_ACTIONS

DEFINE_STATE_ACTIONS(4)
     WHEN ( talking(B, A) )
          DO ( GOTO (2); )
     WHEN ( talking(A, B) && T > 15 )
          DO ( GOTO (1); )
END_STATE_ACTIONS.
```

We have given a flavor of how well suited the situation calculus is to the task of camera control. It is worth pointing out that other actions which require notions of continuous actions, such as panning, can also be handled without sacrificing our discrete model of the world. We can imagine an action that pans a camera a small amount in a certain direction. Then a "continuous" panning action can be defined as a complex action that successively moves the camera in that direction while some condition remains satisfied.

Chapter 5

Physics-based Applications

In this chapter we consider the problem of how to adapt our approach to a much more complex environment. In particular we consider an underwater physics-based world. The main difficulty that presents itself is the unpredictability of the world. That is, we do not wish to axiomatize all the underlying equations used in the physical simulation, nor do we wish to axiomatize the decision making processes of all the characters within the world. Our reticence stems not from lethargy but from concerns of realism and practicality. That is, it is unrealistic and infeasible to allow our characters to be able to exactly predict all events in their world. Moreover, we would incur an unacceptable performance penalty. Our solution is to have our characters represent their desires and intermittently update their world knowledge with sensors.

Our approach is demonstrated with an implementation that we refer to as a *character design workbench*, CDW. The reader is referred back to chapter 1, where figure 1.3 gives an overview of CDW. It consists of a low-level reactive behavior system and a high-level reasoning system, referred to as the *reactive system* and the *reasoning system*, respectively.

5.1 Reactive System

At the foundation of CDW is a low-level reactive behavior system that implements "primitive behaviors" common to all characters. It also autonomously executes and arbitrates among these behaviors. Such behaviors include, but are not necessarily limited to, reactive ones such as "avoiding collisions", and basic locomotive capabilities such as "go to a particular location". Finally, this underlying system is able to return any requested sensory data that it has access to.

It is important to realize that we could implement the primitive behaviors in the high-level reasoning system. Separating them out, however, means that they become less flexible and can not be automated through logical reasoning. There are however a number of benefits that arise from the separation:

- A major strength of the high-level reasoning system lies in the ease with which behaviors specified by it can be reconfigured. Primitive behaviors are typically character-independent, once they are operational the need to change them is minimal.
- Primitive behaviors are intended to be components of high-level behaviors. They should thus be as efficient as possible as they will be executed frequently.
- An independent reactive layer can act as a fail-safe, should the reasoning system temporarily fall through. The reactive layer can thus be used to avoid the character doing anything "stupid" in the event that it can't decide on anything "intelligent". If the reactive layer is too sophisticated and tries to actively pursue it's own high-level goals there will be problems with persistence and stability. Behaviors such as "continue in the same direction, avoiding collisions" is an example of a suitable default reactive system behavior.

- As far as we know real animals do not use reasoning for low-level behaviors. However, what happens in Nature need not constrain how we should approach related problems in computer animation. Nevertheless, Nature does give us confidence that reactive low-level behaviors are one possible solution.
- Finally, from a pragmatic point of view, prior behavioral animation research [114, 23] describe working systems that would suit our purposes.

5.2 Reasoning System

A character's knowledge of its world is referred to as the *background domain knowledge*. It is a set of user supplied axioms that collectively constitute a *causal theory* and that provides the character with an understanding of when actions are possible and how they affect the world. Reasoning with respect to the underlying causal theory is carried out by the reasoning engine, which is implemented as a theorem prover.

The desired, character-dependent behaviors are defined by sketch plans that represent advice to the character on how to behave. These sketch plans enable the inclusion of advice to the character on how to achieve its goals. In addition they can incorporate sensing, thus allowing a character's behavior to be contingent on the current state of the world as it perceives it. The character can follow the sketch plan by using its background domain knowledge and reasoning to infer the missing details. Due to the complex nature of the interactions of the character with its dynamic world, the initial sketch plan may not produce the desired visual results.

As one would expect, at any time a simulation can be temporarily suspended, and the user can ask the character about the state of its world, and what it "thinks" would be the effect of performing various actions. Our system has the advantage that the character can use its reasoning engine to answer extremely precise questions. This goes beyond the simple state queries of regular debuggers. The information from our queries, and the visual feedback from the animations, can then be used to modify the sketch plan. This way of directing a character is analogous to the practice of giving a real actor "notes" after a theatrical performance [49]. This has proved extremely useful, to us at least, as a way of "debugging" high-level behaviors while maintaining the character's complete autonomy. In particular, the character always remembers its new desired behavior and so there is no need to keep correcting the same mistakes.

In the remainder of the chapter, we shall give further details on the reasoning system, the reactive system and how they interact to automatically produce interesting character animations.

5.3 Background Domain Knowledge

The background domain knowledge defines the way a character thinks about its world. Instruction and interaction can be made easier by using representations that correspond to the animator's way of thinking about the character's world. The domain knowledge states when actions are possible, and what the effect of actions are on the world.

The success of our approach rests heavily on our use of the *situation calculus* for representing a character's knowledge. From the user's point of view, the underlying theory can be completely hidden, but for a complete understanding of our approach, the mathematical details are important. The interested reader is referred back to chapter 3 for the details. To reinforce our ideas, we shall employ a simple example of a predator and a prey.

We can represent the prey's position as a fluent PreyPos, where $\mathsf{PreyPos}(p, s)$ means that the prey is in region p in situation s. The predator's position, PredPos, and the prey's desired position, PreyGoalPos, are similarly defined. A property that does not change can just be represented as a relation with no situation argument. For example, $\mathsf{Occluded}(p, q, b)$ states that region p is hidden from region q by obstacle b, and $\mathsf{Occupied}(p, b)$ states that region p is occupied by b.

The reader should not be misled at this point into thinking that we have limited our approach to only two characters. Our example is designed to explain the basic approach. In the implementation we use more scalable naming conventions. That is, we do not have separate fluents for PredPos and PreyPos; we just have a

fluent $\mathsf{Position}(i, p, s)$ that gives character i's position. Then we have other predicates to specify the type of a character i, such as: $\mathsf{Type}(\text{jaws}, \text{predator})$, and $\mathsf{Type}(\text{duffy}, \text{prey})$. We should also point out that the truth of predicates such as $\mathsf{Occupied}(p, b)$ can be quickly calculated at run-time; it need not (and should not) be stored in an unwieldy precomputed database.

5.4 Phenomenology

A key challenge that faces at this point is what do we do next? Intuitively, we should like to start writing down axioms to describe how various actions affect the prey's position, say. The physics-based nature of our current application means that such a simple minded approach is doomed to failure. Consider the action of "turning left". As we shall see in section 5.8, the state of the world after executing this action is an extremely complicated function of muscle activation functions, water force reaction, obstacle location, etc. These are precisely the sort of details that we prefer to (and should) leave out of the high-level behavior controller. If a person wants to reason about wanting to go to a shop, the planning activity takes place at a fairly high-level. It is not normal to precisely fix a path, taking into account wind speed, blood sugar levels, etc. The key observation here is that we want to deal with a character's *intentions*. What actually occurs is highly unpredictable and our best hope would be to fix a course of action and then update it in light of any knew information our senses might provide us about the world.

There has been much work on logics of intention and belief. Unfortunately, the details of how all this relates to the situation calculus are still being worked out (see [100] for some preliminary work). In the meantime we propose the following pragmatism: We will only view certain fluents as representing a character's intentions at the meta-level. That is, there will be no way, within our language, of determining that a fluent pertains to the world, or to the character's mental state. Put more bluntly, we simply consider a character's brain to be part of the world. The price we pay for this simplification is that we lose the ability to talk about goals and intentions *within* our language.

As an example, let us introduce a fluent $\mathsf{TryHide}(s)$, that is true if the prey is trying to hide. We view the fact that the prey is trying to hide as a fact about the world, rather than an esoteric statement about the prey's mental state. This is not as peculiar as it might first appear. In the real world a person's inner thoughts might be viewed as somewhat opaque. In our virtual world any such distinction is clearly arbitrary. There is nothing inherently different about peering into our character's heads to see what their current intentions are to looking up the current rate of flow of the virtual water. Both really are just facts about the virtual world.

Having said all this, problems remain. In particular the question arises about what to do with fluents that pertain to the character's view of its world. For example, the fluent $\mathsf{predPos}$ could legitimately refer to the predator's position, or the position the prey "thinks" the predator is in. We choose the former, the latter concept is represented as an interval-valued fluent (see section 3.10) and the correspondence between the two maintained by sensing. We shall return to this issue in section 5.4.1.

For now let us introduce an action $\mathsf{setGoal}(g)$ that sets the prey's goal position to region g. We can specify an action precondition axiom that gives necessary and sufficient conditions for when the action $\mathsf{setGoal}(g)$ is possible (note that as always unbound variables are implicitly assumed to be universally quantified):

$$\mathit{Poss}\,(\mathsf{setGoal}(g), s) \quad \Leftrightarrow \quad \neg\exists c\ \mathsf{Occupied}(g, c),$$

i.e., $\mathsf{setGoal}(g)$ is possible if, and only if, region g is not occupied. Let us now describe the effect of the action on the fluent $\mathsf{TryHide}(s)$ by stating action effect axioms. In particular, we have a positive effect axiom:

$$\mathit{Poss}\,(\mathsf{setGoal}(g), s) \wedge \mathsf{PredPos}(q, s) \wedge \exists b\ \mathsf{Occluded}(g, q, b) \quad \Rightarrow \quad \mathsf{TryHide}(\mathit{do}(\mathsf{setGoal}(g), s)),$$

which states that provided it is possible, the prey is trying to hide if it is going to a region that is hidden from the predator.

We also have corresponding a negative effect axiom:

$$\mathit{Poss}\,(\mathsf{setGoal}(g), s) \wedge \mathsf{PredPos}(q, s) \wedge \neg\exists b\ \mathsf{Occluded}(g, q, b) \quad \Rightarrow \quad \neg\mathsf{TryHide}(\mathit{do}(\mathsf{setGoal}(g), s)),$$

which states that the prey is not trying to hide if it is going to a non-hidden region.

As we saw in chapter 3 we can now (by simple syntactic manipulation) replace the effect axioms with successor state axioms that express necessary and sufficient conditions for a fluent to change its value. For example, the successor state axiom for the fluent $\mathsf{TryHide}(s)$ states that the prey is trying to hide if, and only if, the goal position just changed to a hidden region, or it was trying to hide in the previous situation and things did not change to make the goal position visible:

$$\begin{aligned} \mathit{Poss}\,(a,s) \Rightarrow \Big[& \mathsf{TryHide}[\mathit{do}(a,s)) \Leftrightarrow \\ & \{a = \mathsf{setGoal}(g) \wedge \mathsf{PredPos}(q,s) \wedge \exists b\ \mathsf{Occluded}(g,q,b)\} \vee \\ & [\mathsf{TryHide}(s) \wedge \{\neg\exists g\ (a = \mathsf{setGoal}(g) \wedge \mathsf{PredPos}(q,s) \wedge \neg\exists b\ \mathsf{Occluded}(g,q,b))\}] \Big] \end{aligned}$$

As we saw in section 3.6 the actions we have defined so far are termed primitive actions. This term comes from the cognitive robotics literature. It is somewhat misleading in this context, as some primitive actions, such as "go to position g", may entail complicated obstacle avoidance behavior in the reactive system.

5.4.1 Incorporating Perception

Back in chapter 1, section 1.4 we discussed the need for perception. We used an example of a falling stack of bricks to explain how sensing was important in order to avoid complicated, inefficient and unnatural behavior. With this chapter's emphasis on physics-based applications one further observation is worthwhile. That is, even if we do not want to exactly pre-compute everything, some pre-computation may be a good idea. To return to the brick example, it would be quite foolish for our character to simply stand directly underneath the falling bricks with an air of blasé sang-froid. Clearly, the character should retire to a safe distance, whence it might await the outcome unafeared. Attempting to perform approximate calculations based on physical laws is often referred to as *qualitative physics*. In [60] an example is given on how to discuss physical processes within the situation calculus. While we relegate the adoption of such ideas to future work, this is a good juncture to point out their efficacy.

Rolling forward

Consider the maze example introduced in chapter 4, section 4.4. In the example the character, in effect, had a map of the maze that it used to plan its path before ever setting foot in the maze. The character plans its path by searching the map and backtracking when it gets stuck. It only executes the plan once it has finished finding a path. Now suppose we have some monsters wandering through the maze in a manner that is unknown to the character. The character can still plan a path but must also keep a look out for monsters. The important point is that the monsters are not marked on the map and so the character can only check for monsters once it is inside the maze. Therefore, the scenario we envisage is that a character executes some actions then looks around to observe any of the unpredictable effects. For example, the character might go to a point in the maze and then look to see if it can see any monsters approaching. Thus sensing necessarily entails executing the actions the character has decided upon up until that point. That is, the character can not look for monsters until it has executed the actions that place it inside the maze.

Another important point is that when a character is planning an action it can change its mind without influencing its world. This is no longer, necessarily, the case once an action has been executed. For example, a character may see that a path leads to a dead end and may return from whence it came, but, in the meantime it may have been spotted by a monster that now begins to pursue it.

Executing a sequence of actions is known as "rolling forward". To see how it works suppose a character is considering some sequence of actions $a_0, \ldots, a_n$. After executing such a sequence of actions the character will be in a situation $\mathit{do}(a_n, \cdots, \mathit{do}(a_1, \mathrm{s}_0) \cdots)$. The successor state axioms allow the character to determine the value of various fluents in the situation s'. Depending on the value of these fluents the character may want to choose an alternative sequence of actions. At some point the sequence may become unwieldy, or an action (such as sensing) may be required that forces the character to commit to the current sequence. We capture the notion that, after executing them, the actions are no longer subject to change by specifying

a new initial situation. Then we can discard the old initial situation and proceed with respect to the new situation. So for each fluent f, we have that $f(\mathrm{s}_0^{\mathrm{new}})) = y$ if and only if $f(do(a_n, \cdots, do(a_1, \mathrm{s}_0^{\mathrm{old}})) = y$.

Sensing

We now return to the issue of sensing. The approach we take is as detailed in section 3.10. That is, we simply employ IVE fluents. To see how this works in our current example, imagine that we define a fluent $\mathsf{Time}(t, s)$ that gives the time t is situation s. The time is updated by tick actions that, for now, we assume to be a fixed distance apart (ten frames in our current implementation). At every tick the reasoning system commits to the actions it has decided upon in the previous time period. This involves sending the "primitive" actions to the underlying reactive system and "rolling" forward the state.

The underlying reactive behavior system then executes the primitive actions and returns some sensory information. The new sensory information is used to update all the IVE fluents for which we obtained information about their new values. The remaining IVE fluents' intervals are expanded to include any possible value that they could have at the current time.

As an example consider a fluent $\mathcal{I}_{\mathsf{brightnessLevel}}(s)$, and an action senseHowBright, then we have two effect axioms:

$$\begin{aligned} \mathsf{BrightnessSensorResponding}(s) &\Rightarrow \mathcal{I}_{\mathsf{brightnessLevel}}(do(\mathsf{senseHowBright}, s)) = \\ &\qquad \langle \mathsf{brightnessLevel}(s), \mathsf{brightnessLevel}(s) \rangle \\ \neg\mathsf{BrightnessSensorResponding}(s) &\Rightarrow \mathcal{I}_{\mathsf{brightnessLevel}}(do(\mathsf{senseHowBright}, s)) = \langle 0, \infty \rangle. \end{aligned}$$

So, if the brightness sensor is not responding, we don't know what the brightness level is. If we know the maximum rate of change of the brightness sensor, then we can make a tighter interval with some simple computation based on the time increment.

5.4.2 Exogenous actions

For each IVE fluent $\mathcal{I}_f$, there will be a corresponding fluent f. These fluents values are updated by exogenous actions (see section 3.7). For example, consider the fluent predPos. There is an exogenous action $\mathsf{movePred}(\boldsymbol{x})$, such that after executing the action the predators new position will be $\boldsymbol{x}$. The exogenous actions are generated by modifying the definition for complex actions. For example, the rule for primitive actions (given in appendix D) is:

$$Do(\alpha, s, s') \triangleq Poss(\alpha, s) \wedge s' = do(\alpha, s) \quad \alpha \text{ is a primitive action}$$

This is modified to be:

$$\begin{aligned} Do(\alpha, s, s') &\triangleq Poss(\alpha, s) \wedge s' = do(\alpha, s) \quad \alpha \text{ is a primitive action and } \alpha \neq \mathsf{tick} \\ Do(\mathsf{tick}, s, s') &\triangleq Poss(\mathsf{tick}, s) \wedge s' = do(\mathsf{movePred}(\boldsymbol{x}), do(\mathsf{tick}, s)) \end{aligned}$$

Similarly we can add an arbitrary number of exogenous actions after various primitive actions. Thus we can avoid having to formalize all aspects of the domain.

5.5 Advice through "Sketch Plans"

We can give advice to a character by building up *complex actions* from the available primitive ones. Complex actions were first described back in chapter 3, section 3.6. Let us assume some new primitive actions and fluents whose intended meaning is given, informally, by their names. Then the character that tries to hide

from predators if it can, and otherwise tries to run away, can be defined by the "program":

proc evade
  while PredatorApproaching **do**
    sense ⨟
    update ⨟
    if $\exists r$ Occupied$(r) \wedge$ Nearby(r) **then**
      (πr)(Occupied$(r) \wedge$ Nearby$(r))?$ ⨟ hideBehind(r)
    else
      (πr)(AwayPredDir$(r))?$ ⨟ setGoal(r)
  od
end

Notice that the above specification may admit more than one candidate as an obstacle to hide behind. We can simply leave it up to the character's reasoning system to select a suitable one at run time. Hence, the character's reasoning engine can be used, in a focussed way, to fill in details that are otherwise tedious to specify. More importantly, as the director of virtual characters, the animator can control the amount of work that should be left to the character. This is a major strength of our approach: we can experiment with various behavioral control strategies with great ease by placing heavy reliance on our character's reasoning abilities. This reduces the amount of work required from the animator and is hence extremely useful for rapidly developing new characters. When high execution speed is required, for example in computer games, we may be able to make our controllers more efficient by gradually reducing the non-determinism. Furthermore, the control structures in our language closely resemble those of a conventional programming language. Apart from making our language easy to use, this allows fast and easy translation to a conventional imperative language for a production version of our controller that will, say, run on a slower machine.

5.6 Implementation

The reasoning engine of CDW was written using the Quintus Prolog Development System in a combination of C and Prolog. For additional speed, the macro expansion of the high-level control commands is written in C (see appendix E). The truth of the fluents in the resulting situations is determined by the Prolog part. The background domain knowledge and the high-level control specification are read into the reasoning engine. The macros are then expanded to generate queries that are answered by the underlying Prolog theorem prover.

Developing controllers using CDW combines the ease of logic programming with the potential efficiency of imperative programming. It is like having a regular imperative programming language with a built in theorem prover to shoulder some of the work. In the initial stages of development, this is extremely useful in quickly getting working prototypes.

To make these ideas more concrete, we shall give some examples taken directly from the source files of our controllers. Consider the code we wrote to help prey avoid being eaten. Our first attempt was to simply divide the scene into a regular two-dimensional grid. We could then enumerate all our regions very simply with the predicate `fl_region(R)`. We defined a fluent `fl_evaluator(E,I,S)`, which, for each situation `S`, and character `I`, maintained a function `E` used for evaluating the regions. Finding suitable hiding positions amounted to nothing more than performing an action `act_evalRegions(I)`. This action updated a fluent `fl_bestGoalPosns(I,GS,S)`, that maintained a list of the best goal positions `GS`. The action `act_pickGoal` then set the character's current intention to any of the current best goal positions. The control program was

thus:

```
proc(control_agent(I),
  act_sense(I) :
  act_checkAlive(I) :
  (
    act_updateMemory(I) :
    act_setEvaluator(I,obstacles) :
    act_evalRegions(I) :
    act_pickGoal(I)
  ) |
  no_op
).
```

The controller relies on the precondition of `act_updateMemory(I)` including the fact that character `I` is still alive. If the precondition is false the controller will not waste its time deciding what to do. The controller got us up and running, but with many characters or three-dimensions there would be too many regions to look at for the approach to be scalable. As one would expect from a logic programming language, we were quickly able to reconfigure and extend our controller by defining a function to randomly sample the space around a point in concentric circles. The radius of the region was given as a fluent `fl_searchRegion(I,EXT,S)`.

```
proc(control_searchForGoalPosn(I),
  act_setEvaluator(I,searchRegions) :
  act_setAcceptable(I,searchRegions) :
  (
    act_expandSearchRegion(I) :
    act_evalRegions(I)
  )*
).
```

Note how we specify to expand the search region zero or more times. We cannot expand indefinitely because the precondition for `act_expandSearchRegion(I)` includes an upper bound.

We then incorporated this controller into the previous one.

```
proc(control_agent(I),
  act_sense(I) :
  act_checkAlive(I) :
  (
    act_updateMemory(I) :
    (
      control_testCurrPosn(I) |
      control_testCurrGoalPosn(I) |
      control_searchForGoalPosn(I) |
      control_testObstacles(I) |
      act_panic(I)
    ) :
    act_pickGoal(I)
  ) |
  no_op
).
```

This controller again relies on the fact of the character being alive being among the preconditions of `act_updateMemory(I)`. It also uses the preconditions for `act_pickGoal(I)`. In particular, we have as a precondition to `act_pickGoal` that a good enough goal must have been found. What constitutes "good enough" is maintained by the fluent `fl_acceptable(A,S)`. This meant the search could stop as soon as we found a good enough position. Therefore, the controller tries out the current position and stops immediately if it is good enough. Otherwise it backtracks and tries the other options. Note that one of the options is

`control_searchForGoalPosn(I)`. If we look back at the definition of `control_searchForGoalPosn` we see how we gradually expand the search region. The preconditions to `act_pickGoal` ensure that we will always stop as soon as possible. Aside from expanding the search region to its maximum, the controller can also consider the obstacles it had seen on its travels. This is on the assumption that there will be hiding places near obstacles. Finally, if all else fails it can panic. Panicking causes the acceptability criterion, as given by `fl_acceptable(A,S)`, to be ignored and the best position found so far wil be chosen. Once again we see how the nondeterminism in our specifications simplifies writing down controllers.

With the new controller, we were able to control multiple characters in three-dimensions. A more complete listing of the code is given in appendix F. Although there is still much room for improvement we have demonstrated a methodology for developing efficient controllers. With five characters we can, on average, manage about five frames per second, on a SGI Indigo Impact workstation, using wireframe rendering, and running Character Design Workbench.

5.7 Correctness

It is an old adage of computer science that, while important, testing a program can only ever prove the presence of bugs, never their absence. Therefore, one potentially significant aspect of using the situation calculus is the ability to prove properties of our specifications. One could envisage proving the presence or absence of certain (un)desirable traits in the specification of a character's behavior. Moreover, this can be done early on in the software life-cycle, potentially eliminating bugs that would otherwise be costly to rectify later on.

For example, in section 5.9 we discuss an animation entitled "Pet Protection". For this animation the hero of the piece had a controller based on a specification fragment as follows:

if $\neg \mathsf{InDanger}(x) \wedge \mathsf{InDanger}(y) \wedge \mathsf{isPet}(x, y)$ **then**
  $\mathsf{rescue}(x, y)$
else
  $\mathsf{runAway}(x)$
fi

Now suppose we have a couple of characters pet, and hero such that pet is hero's pet, $\mathsf{IsPet}(\mathsf{hero}, \mathsf{pet})$. Now suppose, in some situation s, we have that the pet is in danger $\mathsf{InDanger}(\mathsf{pet}, s)$, but that the hero is also in danger, $\mathsf{InDanger}(\mathsf{hero}, s)$. Then, it is possible to prove that in such a situation the hero will not go to the rescue but will run away. The proof just involves some simple rules of logic and expanding out the specification according to the rules given in appendix D.

$$
\begin{array}{l}
\mathit{Do}\,(\mathbf{if}\ \neg\mathsf{InDanger}(x) \wedge \mathsf{InDanger}(y) \wedge \mathsf{IsPet}(x,y)\ \mathbf{then} \\
\quad \mathsf{rescue}(x,y)\ \mathbf{else}\ \mathsf{runAway}(x)\ \mathbf{fi}, s, s') \\
\triangleq \\
\mathit{Do}\,[((\neg\mathsf{InDanger}(x) \wedge \mathsf{InDanger}(y) \wedge \mathsf{IsPet}(x,y))?\ ;\ \mathsf{rescue}(x,y)) \mid \\
(\neg(\neg\mathsf{InDanger}(x) \wedge \mathsf{InDanger}(y) \wedge \mathsf{IsPet}(x,y))?\ ;\ \mathsf{runAway}(x)), s, s'] \\
\triangleq \\
\mathit{Do}\,((\neg\mathsf{InDanger}(x) \wedge \mathsf{InDanger}(y) \wedge \mathsf{IsPet}(x,y))?\ ;\ \mathsf{rescue}(x,y), s, s') \vee \\
\mathit{Do}\,(\neg(\neg\mathsf{InDanger}(x) \wedge \mathsf{InDanger}(y) \wedge \mathsf{IsPet}(x,y))?\ ;\ \mathsf{runAway}(x), s, s') \\
\triangleq \\
\exists s^*\ [\mathit{Do}\,((\neg\mathsf{InDanger}(x) \wedge \mathsf{InDanger}(y) \wedge \mathsf{IsPet}(x,y))?, s, s^*) \wedge \mathit{Do}\,(\mathsf{rescue}(x,y), s^*, s')] \vee \\
\exists s^*\ [\mathit{Do}\,(\neg(\neg\mathsf{InDanger}(x) \wedge \mathsf{InDanger}(y) \wedge \mathsf{IsPet}(x,y))?, s, s^*) \wedge \mathit{Do}\,(\mathsf{runAway}(x), s^*, s')] \\
\triangleq \\
\exists s^*\ [\neg\mathsf{InDanger}(x,s) \wedge \mathsf{InDanger}(y,s) \wedge \mathsf{IsPet}(x,y) \wedge s^* = s \wedge \mathit{Do}\,(\mathsf{rescue}(x,y), s^*, s')] \vee \\
\exists s^*\ [\neg(\neg\mathsf{InDanger}(x,s) \wedge \mathsf{InDanger}(y,s) \wedge \mathsf{IsPet}(x,y)) \wedge s^* = s \wedge \mathit{Do}\,(\mathsf{runAway}(x), s^*, s')] \\
\Rightarrow
\end{array}
$$

$$\exists s^* \ [\neg \mathsf{InDanger}(x, s) \wedge \mathsf{InDanger}(y, s) \wedge \mathsf{IsPet}(x, y) \wedge s^* = s \wedge \mathit{Do}\,(\mathsf{rescue}(x, y), s^*, s')] \vee$$
$$\exists s^* \ [\mathsf{InDanger}(x, s) \vee \neg \mathsf{InDanger}(y, s) \vee \neg \mathsf{IsPet}(x, y) \wedge s^* = s \wedge \mathit{Do}\,(\mathsf{runAway}(x), s^*, s')]$$
$$\Rightarrow$$
$$[\neg \mathsf{InDanger}(x, s) \wedge \mathsf{InDanger}(y, s) \wedge \mathsf{IsPet}(x, y) \wedge \mathit{Do}\,(\mathsf{rescue}(x, y), s, s')] \vee$$
$$[\mathsf{InDanger}(x, s) \vee \neg \mathsf{InDanger}(y, s) \vee \neg \mathsf{IsPet}(x, y) \wedge \mathit{Do}\,(\mathsf{runAway}(x), s, s')]$$
$$\Rightarrow$$
$$[\neg \mathsf{InDanger}(\mathsf{hero}, s) \wedge \mathsf{InDanger}(\mathsf{pet}, s) \wedge \mathsf{IsPet}(\mathsf{hero}, \mathsf{pet}) \wedge \mathit{Do}\,(\mathsf{rescue}(\mathsf{hero}, \mathsf{pet}), s, s')] \vee$$
$$[\mathsf{InDanger}(\mathsf{hero}, s) \vee \neg \mathsf{InDanger}(\mathsf{pet}, s) \vee \neg \mathsf{IsPet}(\mathsf{hero}, \mathsf{pet}) \wedge \mathit{Do}\,(\mathsf{runAway}(\mathsf{hero}), s, s')]$$
$$\Rightarrow \qquad \text{(by assumption that } \mathsf{InDanger}(\mathsf{hero}, s)\text{)}$$
$$\mathit{Do}\,(\mathsf{runAway}(\mathsf{hero}), s, s')$$
$$\triangleq$$
$$s' = \mathit{do}(\mathsf{runAway}(\mathsf{hero}), s)$$

In general such proofs are best conducted by a computer program as they mainly involve lots of substituting definitions. Indeed this is exactly the process the interpreter goes through when calculating the appropriate behavior.

Of course, the extent of what we can prove is somewhat limited in our current setting. We have already discussed at some length the undesirability of axiomatizing all the laws of physics that pertain to our undersea simulation. Without such an axiomatization, however, we can not prove, for example, that a predator will capture a prey from a given an initial configuration. The most we could hope for is a proof that it will always *try* to capture a prey in a given situation. This opens up avenues for future research to try and improve on this. In particular we might consider using qualitative physics and automated proof assistants to enable us prove stronger results.

5.7.1 Visibility Testing

One important aspect of our pursuit and evasion example was the ability of the prey to locate hidden regions. The task is nontrivial and a bad solution could result in unacceptably slow execution times. The first observation to make is that, if we think of the predators as light sources, the problem of determining regions that are hidden from them is closely related to the rendering problem of fast shadow computation. In our case, the problem is exacerbated by the fact that our "light sources" are moving around.

One possibility is to use OpenGL® and take advantage of specialized graphics hardware to provide a fast solution to the problem. The solution we take however, is based on octrees [40]. That is, we can bound the whole scene by a cube. The cube can then be recursively subdivided into eight regions to form a tree like structure. In theory, whenever we wish to determine the regions hidden from a predator, we check to see which cell the predator is in and then see which cells are obscured by obstacles from that cell. The idea is that by starting at the top of the octree and working down we can quickly discard large regions of space as completely visible or completely hidden.

For example, in figure 5.1, there are no obstacles that interpenetrate the region in between cell A and B. Thus the pair A,B can be marked as completely visible from one another and, for the purposes of comparisons with each other, need not be subdivided further.

In figure 5.2, the obstacle completely occludes cell A and B. Thus the pair A,B can be marked as completely occluded from one another and, for the purposes of comparisons with each other, need not be subdivided further.

In figure 5.3, the obstacle partially occludes cell A and B. Thus the pair A,B must be subdivided further to determine complete visibility information. If the smallest allowable subdivision level has been reached then we decide visibility by polling for visibility at each of the vertices. So in the figure the pair A,B would be marked as visible.

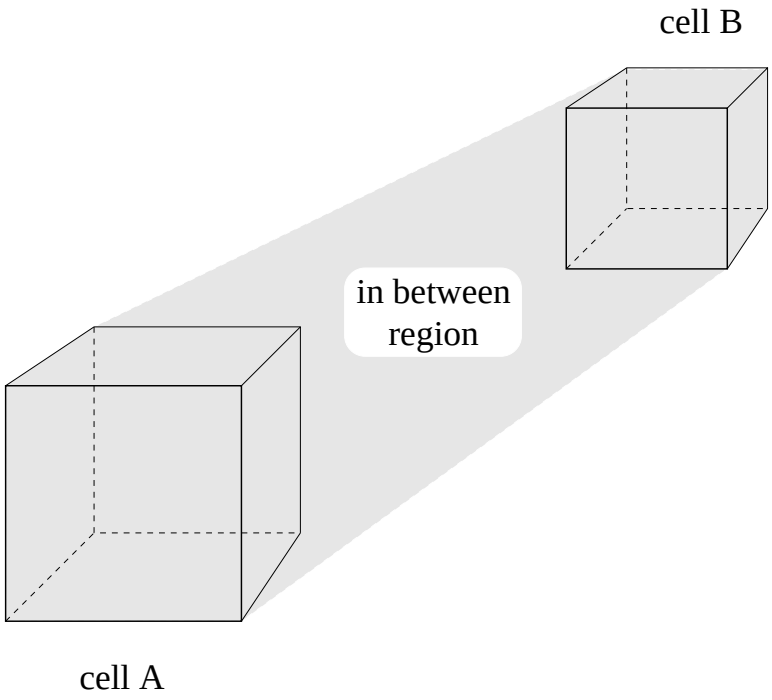


Figure 5.1: Cell A and B are "completely visible" from one another

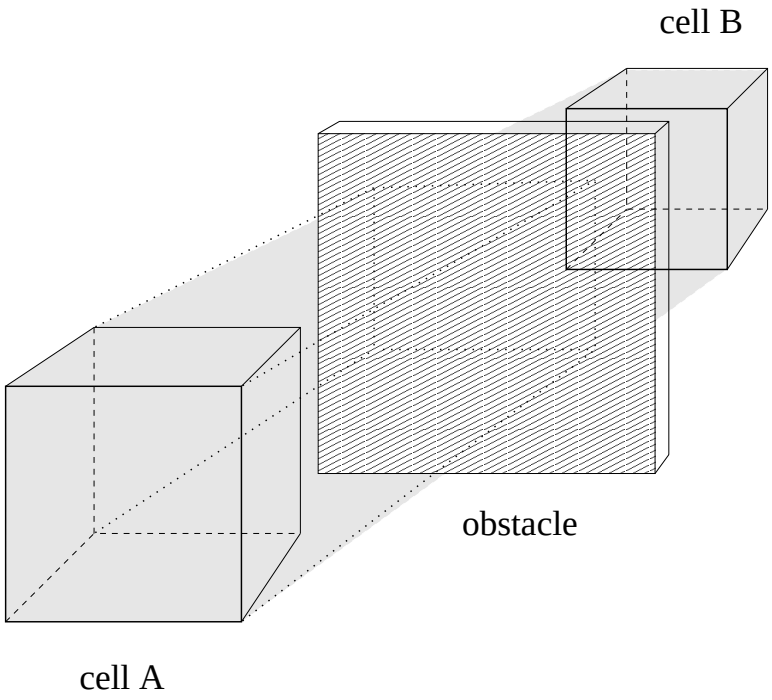


Figure 5.2: Cell A and B are "completely occluded" from one another

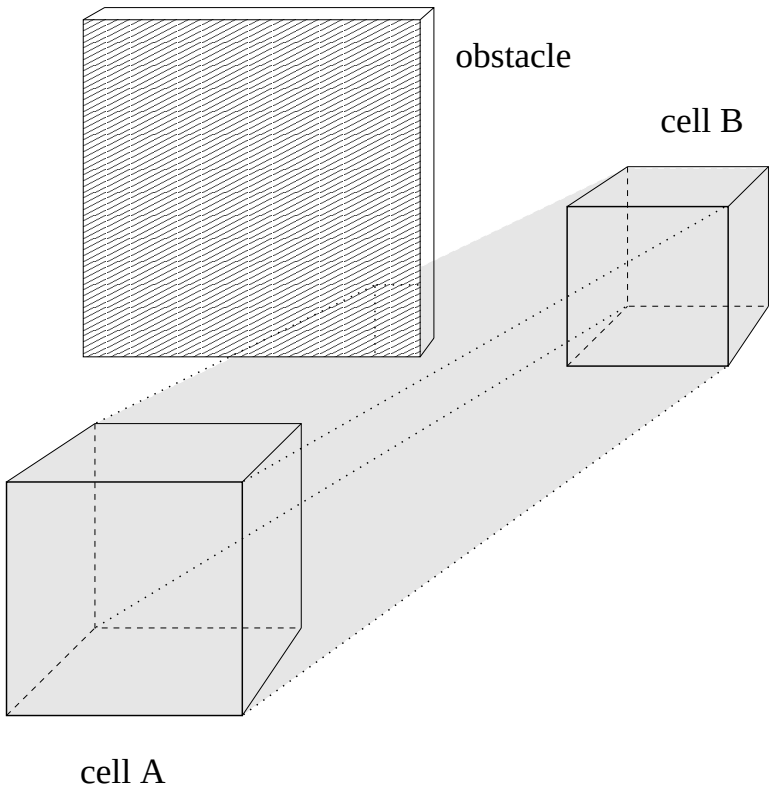


Figure 5.3: Cell A and B are "partially occluded" from one another

In practice, subdividing the whole space down to the required fidelity and computing visibility information for all pairs of cells is too expensive. The solution is to make the following observations:

- Characters, especially those in water, will have a limited visual range. They will also have a limited field of view. Therefore there is no need to compute visibility information for cells beyond a distance, or angle.
- Characters can be given some simple heuristics for finding hidden locations. For example, they can be told that they should only look for cover near obstacles. That is, even if a character is momentarily hidden from a predator when it is open water, such a location is still far from ideal. It is much safer to seek refuge near obstacles. Thus, visibility information need only be computed within the vicinity of obstacles.
- Approximate visibility information can still be calculated by projecting character locations onto the boundaries of intervening regions for which information is available. By combining these tests with additional simple bounding boxline intersection tests, the reliability of the approximate visibility tests can be enhanced.
- Characters may well only visit small portions of the scene within an animation. It would be wise to provide a coarse evaluation for all obstacles in the scene. To avoid doing lots of unnecessary work, however, we need only compute high fidelity visibility information for regions the characters visit. The calculations can then be cached in a globally accessible database for future reference by any character.
- It is not catastrophic if the odd mistake is made. It enhances realism and the behavior algorithms should be robust enough to avoid disaster if information is unavailable for short periods of time.

Figure 5.4 shows a two-dimensional version of the space partitioning approach we take to visibility testing. Note that we use bounding boxes for the obstacles. This results in "mistakes", as some of the cells which are

visible are marked as hidden. A solution to this would be to compute tighter bounding boxes. In general, this would entail computing a hierarchical octree of bounding boxes for each obstacle. In practice the flaw has little effect since being hidden is only one of the criterion that the prey use to select suitable goal positions. That is, the erroneously marked cells are on the periphery of the hidden region and as such the other criteria (such as distance from the predators, distance from the predators' predicted positions, size of the hidden region, whether there are hidden regions surrounding the selected region, Etc.) precludes them from being selected as ideal goal positions in most circumstances. When the prey are in open water far from any hidden regions they may be selected for short periods of time until more suitable cells come into range. Again, this is harmless because it does not affect the general observed evasion strategy. Finally, making mistakes for borderline cases is extremely realistic.

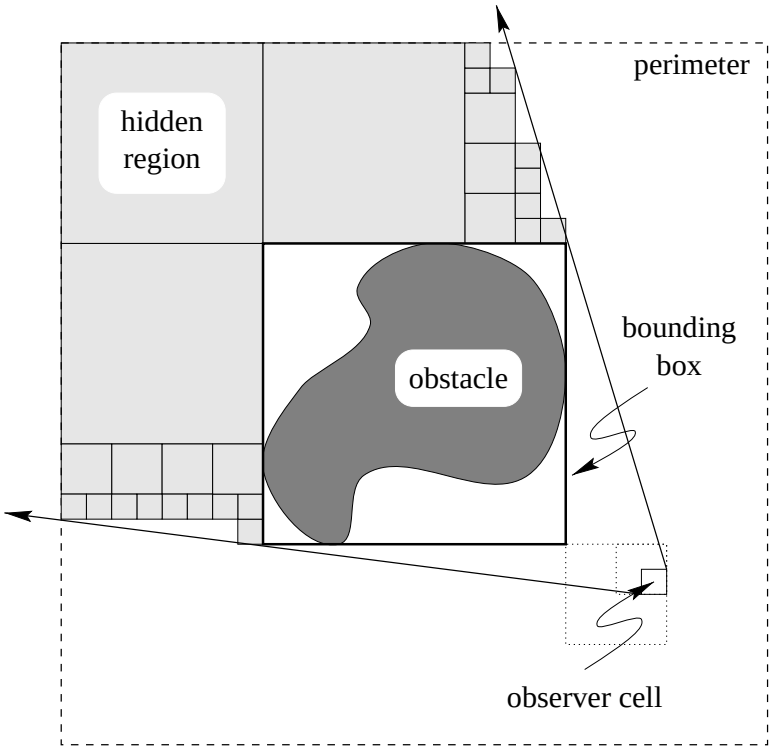


Figure 5.4: Visibility testing near an obstacle.

5.8 Reactive System Implementation

We shall give an overview of the underlying reactive system. The system consists of a number of subsystems. It derives from the work described in [113], to which the reader is referred for further details. The notable enhancements we have made are the ability to import arbitrary geometric models, and a more general collision avoidance mechanism. We shall discuss both of these enhancements. For the sake of completeness, we also include some brief descriptions of the parts that remain unchanged (see [113] for the details).

5.8.1 Appearance

The rendering sub-system system allows us to capture the form and appearance of a merperson. We use texture mapped, three-dimensional geometric display models with which to "envelope" the dynamics model described in section 5.8.2.

3D Geometric Models

Geometric models for our system can be constructed using the Alias|Wavefront^{TM} Studio modeler [2] and automatically imported. Aside from the background scenery, we have created a merman that consists of 2 unique NURBS surfaces: the body (including the tail); and 16 unique polygonal surfaces: the head, upper arms, lower arms, hands, thumbs and ears, as shown in figure 5.5. Together they form an articulated figure arranged in a hierarchical tree structure rooted at the merperson's local coordinate system.

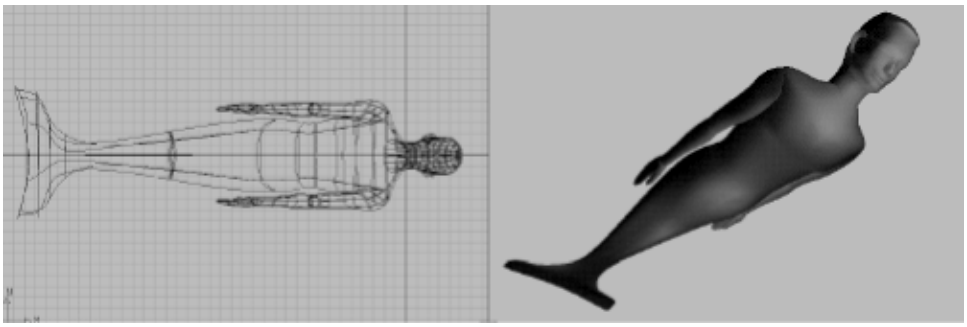


Figure 5.5: The geometric model.

The geometric model provides the appearance of the merperson. In order to make it move, an underlying dynamic model is created. The dynamic model (see section 5.8.2) can be viewed as the "flesh" that moves and deforms over time to produce the locomotion of the merperson. Only the body and tail of the dynamic model actually deform as the merperson swims, the head and limbs move but do not deform. The geometric surfaces are coupled to the underlying dynamic model so that they will move and deform accordingly. This is achieved by associating their control points with the faces of the dynamic model. This is discussed, but not fully implemented in [113].

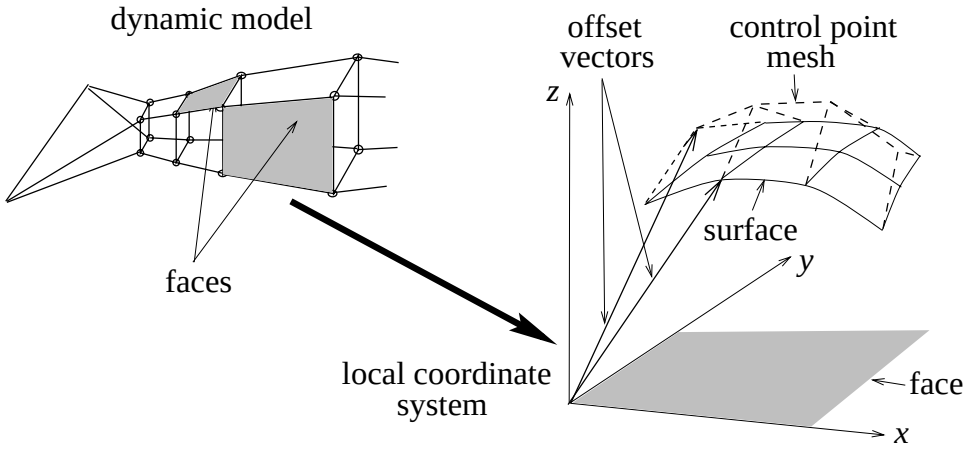


Figure 5.6: Coupling the geometric and dynamic model.

For each face in the dynamic model there is a local coordinate system. Each point in the control point mesh (the dotted lines) is then assigned to the nearest local coordinate system and, in the rest state, an offset vector is calculated (two example offset vectors are shown in the figure). The offset vector is then used to update the control point positions as the underlying dynamic model moves. In practice it is necessary to subdivide the faces of the dynamic model into a 4 by 4 grid of patches, each having its own local coordinate system. In this way artifacts that occur with texture mapping, when the offset vectors inter-penetrate, are minimized.

Texture Mapping

The next step is to map images onto the geometric display models (texture mapping). We painstakingly created the textures using Adobe® Photoshop® using scanned in photographs as our source. The most important step in the texture mapping process is to derive texture coordinates. The texture coordinates map the digital images of different parts of a merperson onto the corresponding 3D surface. To obtain the texture coordinates we wrote software to stretch the irregular shaped images out into rectangular images.

Once the texture coordinates are determined, the rest of the texture mapping procedure can be carried out via a simple function call in any commercially available 3D graphics software package that supports texture mapping. Currently we use OpenGL®, which gives interactive animation rates in wire-frame and shaded modes and, for fully textured rendering, gives about 5fps, on an SGI Indigo^2 High Impact.

Figure 5.7: Texture mapped face.

Texture mapping the merperson's face is especially difficult and we are grateful to have been able to use the laser range finder data used in [64] as our source. After some work "fixing up" the model and image the merman was acceptably aesthetically pleasing. Figure 5.7 shows a closeup of the merman's face. Note that it consists of 8 unique polygonal surfaces. Considerable manual effort was required to ensure the smooth blending of the texture maps for each adjoining surface. Currently we have not attempted any facial animation but our system is ideally suited to its incorporation.

5.8.2 Locomotion

The locomotion sub-system consists of a biomechanical model that captures the physical and anatomical structure of the character's body, including its muscle actuators, and simulates its deformation and physical dynamics. An interface to the underlying model is provided by a set of abstract *motor controllers*. The motor controllers are parameterized procedures, each of which is dedicated to carrying out a specific motor function, such as "swim forward", "turn left" or "ascend". They translate natural control parameters such as the forward speed, angle of the turn or angle of ascent into detailed muscle or arm actions. Four frames from an animation of a merperson swimming are shown in figure 5.8.

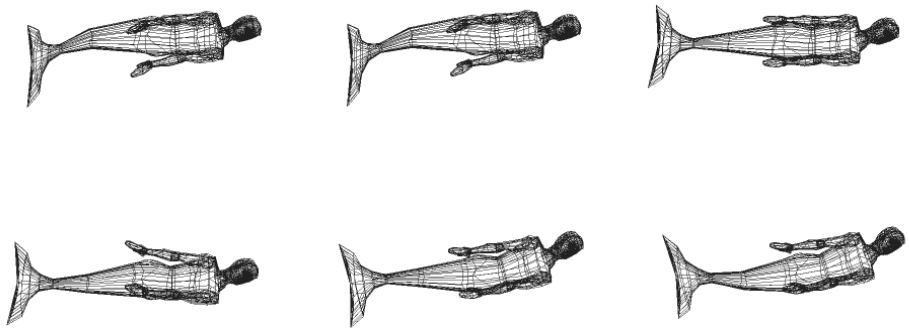


Figure 5.8: A merperson swimming.

The locomotion control problem for physics-based characters is a challenging one. In its full guise the problem involves the locomotion of hierarchical unstable articulated figures subjected to impulsive forces. A lot of progress has been made with producing physically realistic motion for articulated figures. In addition there has been impressive progress with solving the locomotion control problem for creatures that can be modeled as deformable bodies [80, 114, 51]. We choose the latter work for our implementation but the former could also have served.

Deformable models

Currently only the body and tail of the merperson deform. The deformable part is shown in Figure 5.9. It consists of twenty three point mass nodes and ninety one connecting spring and damper units. Each spring and damper unit can deform along one axis. Together they give the body its structure whilst still allowing it to bend. To prevent twisting and shearing each face has two diagonal units. The bold lines depicted in figure 5.9 that span the length of the body are active muscles. In total their are twelve of them and they allow the merperson to deform under its own control. They are arranged in pairs, with two such pairs on each side of the three actuated body segments. The two upper body actuated segments are used for turning whilst the two lower ones are used for swimming.

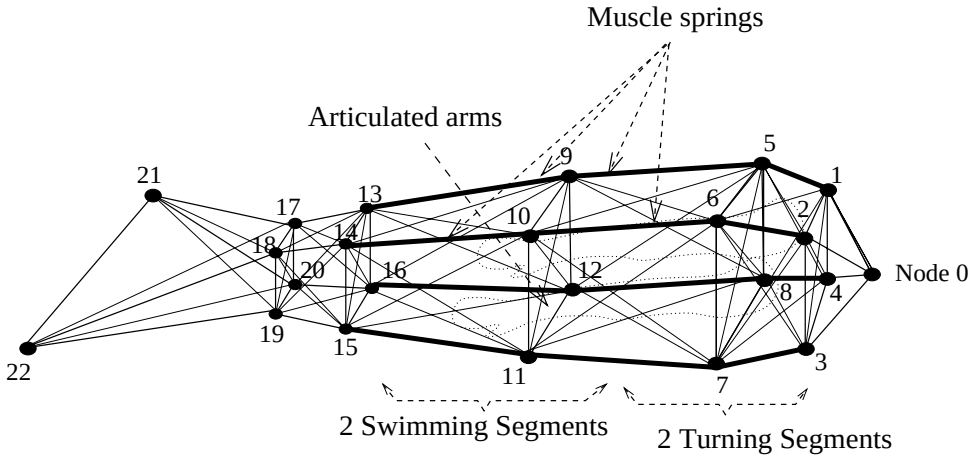


Figure 5.9: The dynamic model.

We shall now employ the nomenclature of chapter 2 to explain how the laws of physics are applied to the mass-spring-damper model. The state vector consists of forty six vector quantities which represent the position and velocity of each of the nodes. All the nodes are governed by the same equations so we shall proceed with a study of a single node i. The mass of node i is denoted by m_i and remains constant. The state vector for one node is the position $\mathbf{x}_i(t) \in \mathbb{R}^3$ and the velocity $\dot{\mathbf{x}}_i(t) \in \mathbb{R}^3$. Our aim is to formulate the state equations that will allow us to calculate the acceleration $\ddot{\mathbf{x}}_i(t) \in \mathbb{R}^3$.

The spring-damper units that connect the nodes are the same as the one depicted back in chapter 2, figure 2.3. We denote the unit that connects node i to node j as s_{ij}, where k^s_{ij} is the associated spring stiffness, and k^d_{ij} the damping factor. The rest length is denoted by p_{ij}. For the active muscles the rest length is a function of time. The activation functions we use were worked out by hand, however, an automatic technique for finding these functions is available in [51].

The vector connecting node i to node j is denoted by $\mathbf{r}_{ij}$, where:

$$\mathbf{r}_{ij}(t) = \mathbf{x}_j(t) - \mathbf{x}_i(t).$$

Thus, $r_{ij}(t) = ||\mathbf{r}_{ij}(t)||$ denotes the length of s_{ij} at time t.

Similarly, the relative velocity of node i with respect to node j is denoted by $\dot{\mathbf{r}}_{ij}$, where:

$$\dot{\mathbf{r}}_{ij}(t) = \dot{\mathbf{x}}_j(t) - \dot{\mathbf{x}}_i(t).$$

Thus, $\dot{r}_{ij}(t) = (\dot{\mathbf{r}}_{ij} \cdot \mathbf{r}_{ij})/r_{ij}$ denotes the normalized relative speed of node i with respect to node j.

The extension, e_{ij}, of s_{ij} the current length minus the rest length:

$$e_{ij}(t) = r_{ij}(t) - p_{ij}(t).$$

The elastic force, $\mathbf{f}_{ij}$ exerted by s_{ij} on node i can then be calculated using Hooke's law:

$$\mathbf{f}_{ij}(t) \quad = \quad \frac{k^s_{ij} e_{ij}(t)}{r_{ij}} \mathbf{r}_{ij} + \frac{k^d_{ij} \dot{r}_{ij}(t)}{r_{ij}} \mathbf{r}_{ij}$$

Note that there is an equal and opposite force $-\mathbf{f}_{ij}(t)$ exerted on node j.

The set of nodes adjacent to node i is denoted N_i. The net internal force exerted on node i due to the spring-damper units can then be obtained by summing up the forces exerted by all the spring-damper units connected to nodes in N_i:

$$\mathbf{f}^s_i = \sum_{j \in N_i} \mathbf{f}_{ij}(t).$$

The other source of force on node i is the water force $\mathbf{f}^w_i$. The water is assumed to be irrotational, incompressible and slightly viscous. We triangulate the faces of the dynamic model. Then, for the sake of efficiency, we approximate the hydrodynamic force on each triangle as

$$\mathbf{f} = \min[0,\ -\mu_w A ||\mathbf{v}|| (\mathbf{n} \cdot \mathbf{v}) \mathbf{n}], \tag{5.1}$$

where μ_w is the viscosity of the water, A is the area of the triangle, $\mathbf{n}$ is its normal, and $\mathbf{v}$ is its velocity relative to the water. The external forces $\mathbf{f}^w_i$ at each of the three nodes of the triangle are incremented by $\mathbf{f}/3$. Therefore, the total force on a node i is:

$$\mathbf{f}_i = \mathbf{f}^w_i - \mathbf{f}^s_i.$$

We are now in a position to give the state equations. They take the form of a set of coupled second-order ordinary differential equations, formulated according to Newton's laws of motion:

$$m_i \ddot{\mathbf{x}}_i(t) = \mathbf{f}_i(t); \qquad i = 0, \ldots, 22, \tag{5.2}$$

To simulate the dynamics of the merperson, the differential equations of motion must be integrated over time. This is made difficult because the system is intrinsically stiff. Indeed there are many common scenarios that may cause the equations to become unstable. For example, executing a right turn to avoid an unexpected collision, say, whilst engaged in a left turn will cause problems. Therefore, to counteract these difficulties, we use a simple, numerically stable, semi-implicit Euler method (see [113] for details).

5.8.3 Articulated Figures

In order to provide increased functionality and realism the merperson has two articulated arms. A researcher may thus begin in the undersea world of non-impulsive forces and then, for a more challenging problem, have the merperson haul itself out of the water, to crawl about on its hands! To date, however, we have concentrated on the movement of the merperson's body instead of the detailed movement of the arms. That is, to simplify the dynamic model and its numerical solution, we do not simulate the elasticity and dynamics of the arms. However, we do approximate the dynamic forces that the arms exert on the body of the merperson to control locomotion.

The articulated arms work by applying reaction forces to nodes in the midsection of the merperson's body, i.e. nodes $N_i, 1 \leq i \leq 12$ (see Fig. 5.9). During swimming the arms are simply used in an analogous way to the airfoils of an airplane. Pitch and yaw control stems from changing their orientations $\pi/4 \leq \gamma \leq \pi$ relative to the body. Assuming that an arm has an area A, surface normal $\mathbf{n}$ and the merperson has a velocity $\mathbf{v}$ relative to the water, the arm force is

$$F_f = -A||\mathbf{v}||(\mathbf{n} \cdot \mathbf{v})\mathbf{n} = -\mathbf{A}(||\mathbf{v}||^{\mathbf{2}} \cos\gamma)\mathbf{n} \tag{5.3}$$

(cf. Eq. 5.1) and is distributed equally to the 6 midsection nodes on the side of the arm. When the arm is angled upward a lift force is imparted on the body and the merperson ascends, and when it is angled down, a downward force is exerted and the merperson descends. When the arm angles differ, the merperson yaws and rolls.

5.8.4 Locomotion Learning

In [51] a learning technique is described that automatically synthesizes realistic locomotion for physics-based models of animals. This technique specifically addresses animals with highly flexible and muscular bodies, such as fish, rays, snakes, and merpeople. In particular, they have established an optimization-based, multi-level learning process that can sit on top of the locomotion subsystem. We initially tried using this system to learn controllers for our creature but it gave unsatisfactory results. In particular, for reasons that we were unable to determine, the merman could not even swim in a straight line! Consequently we resorted to muscle activation functions that were worked out by hand. Regardless, it would be straightforward to incorporate locomotion learning into the underlying reactive system.

5.8.5 Perception

The perception sub-system equips a merperson with a set of "on-board" virtual sensors to provide sensory information about the dynamic environment. It also includes a perceptual attention mechanism which allows the merperson to train its sensors at the world in a task-specific way.

5.8.6 Behavior

The behavior sub-system of the character mediates between the perception sub-system and the motor sub-system. It consists of a behavior arbitrator[1] and a set of behavior routines that implements a repertoire of basic behaviors including "avoiding collisions", "eating", "target following" and "wandering". These primitive behaviors serve a dual purpose. Firstly, they instantiate the "primitive" actions generated by the reasoning system. Secondly, as a whole, they constitute the "default" behavior the character exhibits in the absence of commands ("primitive" actions) from the reasoning system. The fundamental function of the behavior arbitrator is to coordinate the primitive behaviors to generate the default character behavior. Arbitration is done by associating differing priorities with different primitive behaviors. The commands from the reasoning system correspond directly to primitive behaviors and consequently fit elegantly into the behavior arbitration scheme. That is, commands will be executed provided that no more urgent behavior is

[1] In [114] the behavior arbitrator was referred to as an "intention generator", this term may be confusing as its functionality is largely subsumed by the reasoning system.

adjudged to be necessary. For example, a character will head toward a specified location, provided it does not have to avoid a collision. The behavior arbitrator also controls the focuser which returns required sensory data to the reasoning system (see [114] for additional details). At every simulation time step, the behavior arbitrator activates low-level behavior routines that input the filtered sensory information and compute the appropriate motor control parameters to carry the character one step closer to fulfilling the current intention.

In the future we would like to use our behavior specification language to implement the behavior subsystem. We note that the ConGolog language described in [43] would be well-suited to this task.

Some "primitive" actions just update the character's model of its world; the remainder (including all sensing actions) are designated by the user as actions to be communicated to the reactive system. In our current high-level controllers, the reasoning system may select sensing actions, and, for each character, one non-sensing communicable primitive action every ten frames. The reactive system waits up to a specified time limit (currently five seconds) for a non-sensing "primitive" action to be generated. If no such action is forthcoming, it will continue to execute with the low-level default behavior. Any sensing actions generated in the time limit will be processed regardless.

Collision Avoidance

The reactive system already enabled our characters to avoid collisions with cylinders. To improve the generality of the system we choose to implement a potential field approach to collision avoidance.

We based our approach on the equations given in [63]. The idea is that for each point in $\boldsymbol{q} \in \mathbb{R}^3$ we have a force vector $\boldsymbol{F}(\boldsymbol{q}) \in \mathbb{R}^3$ that points the correct direction to travel in order to move toward a goal position whilst avoiding any obstacles. The force vector is defined in terms of a potential function $U : \mathbb{R}^3 \to \mathbb{R}$, such that:

$$\boldsymbol{F}(\mathbf{q}) = -\nabla U(\boldsymbol{q}),$$

where $\nabla U(\boldsymbol{q})$ denotes the gradient of U at $\boldsymbol{q}$.

In general, $U = U_-(\boldsymbol{q}) + U_+(\boldsymbol{q})$, where $U_-(\boldsymbol{q})$ is the *repulsive potential* associated with the obstacles, and $U_+(\boldsymbol{q})$ is the *attractive potential* associated with the goal point. This, in turn, gives us that $\boldsymbol{F} = \boldsymbol{F}_- + \boldsymbol{F}_+$, where $\boldsymbol{F}_- = -\nabla U_-$ and $\boldsymbol{F}_+ = -\nabla U_+$.

To define the repulsive potential, we first define $\rho(\boldsymbol{q})$ to be the minimum distance from the point $\boldsymbol{q}$ to the obstacle. We also define a threshold ρ_0, such that beyond this distance the obstacle has no influence. The repulsive potential we use is now defined, for some constant η, as:

$$U_-(\boldsymbol{q}) = \begin{cases} \frac{1}{2}\eta(\frac{1}{\rho(\boldsymbol{q})} - \frac{1}{\rho_0}) & \text{if } \rho(\boldsymbol{q}) \leqslant \rho_0, \\ 0 & \text{otherwise.} \end{cases}$$

The function is chosen to be differentiable for convex objects.[2] It gives us that the repulsive force is

$$\boldsymbol{F}_-(\boldsymbol{q}) = \begin{cases} \frac{\eta\nabla\rho(\boldsymbol{q})}{\rho^2(\boldsymbol{q})}(\frac{1}{\rho(\boldsymbol{q})} - \frac{1}{\rho_0}) & \text{if } \rho(\boldsymbol{q}) \leqslant \rho_0, \\ 0 & \text{otherwise.} \end{cases}$$

Note that the gradient $\nabla\rho(\boldsymbol{q})$ is the unit vector that points from the closest point on the obstacle toward $\boldsymbol{q}$.

In the case of multiple obstacles we obtain the total repulsive force at a point by summing the repulsive forces from all the obstacles. In our implementation we also clamp the maximum repulsive force. Also, when we calculate repulsive forces we use a bounding box for the obstacles that is deliberately made slightly bigger than necessary. In particular, the box is grown by an amount proportional to the character's size. This increases realism by allowing smaller creatures to get in closer to an obstacle than a large creature. Moreover, since the sharks are larger, this can be exploited in the specification of the evasion behavior for the merpeople. Figure 5.8.6 shows a graphical depiction of the repulsive potential around an obstacle.

[2] We can always decompose concave objects into a set of convex objects.

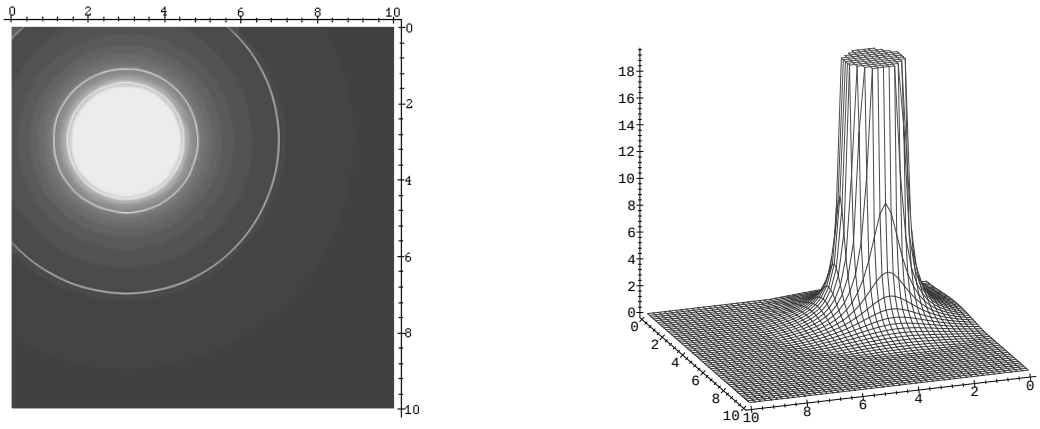


Figure 5.10: The repulsive potential.

The attractive potential has a much simpler equation. First we define $\rho_g(\boldsymbol{q})$ to be the distance from the point $\boldsymbol{q}$ to the current goal position $\boldsymbol{q}$. Then, for some constant ξ, we have that the attractive potential is

$$U_+(\boldsymbol{q}) = \xi\rho_g(\boldsymbol{q}).$$

This gives us that the attractive force is:

$$\boldsymbol{F}_+(\boldsymbol{q}) = -\frac{\xi(\boldsymbol{q}-\boldsymbol{g})}{\rho_g(\boldsymbol{q})}.$$

Note that we must be careful not to divide by 0 at the goal position. Figure 5.8.6 shows a graphical depiction of the attractive potential around a goal position.

Figure 5.8.6 shows a graphical depiction of the superposition of the repulsive and attractive potential fields.

It is the case that the goal position can change as a consequence of a new course of action being decided upon by the reasoning engine. The reactive layer limits the effect that a new goal position may have by imposing a maximum turn angle per time step. In this way the character changes course gracefully, and any momentary oscillation is evened out. For its part the higher level reasoning engine preempts many of the traditional problems with local minima by choosing goals that have a clear path leading to them. The reasoning engine is also able to spot when the merperson is not making progress towards its goal. It can then set a new goal, or, depending on the situation, adjust the parameters to the potential field. This is done by making these parameters fluents which can be changed by certain actions. We believe this synergy of a high-level qualitative system monitoring a low-level numerical procedure is a powerful combination. In general, the potential field is mainly used to add a level of robustness. Any collisions that were unforeseen by the reasoning engine do not result in disaster for the character's well-being.

5.9 Animation Results

Most of our animations to date center around merpeople characters and sharks. The sharks try to eat the merpeople and the merpeople try to use the superior reasoning abilities we give them to avoid such a fate. For the most part, the sharks are instructed to chase merpeople they see. If they can't see any, they go to where they last saw one. If all else fails they start to search systematically.

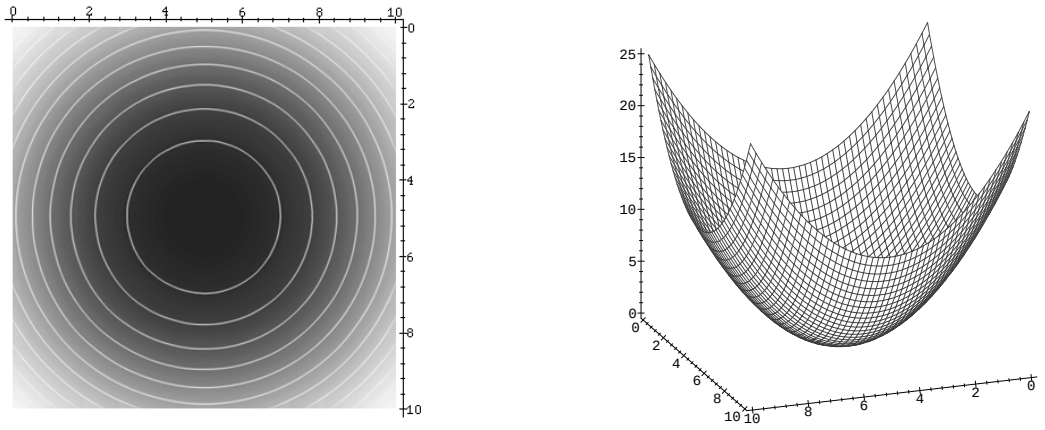


Figure 5.11: The attractive potential.

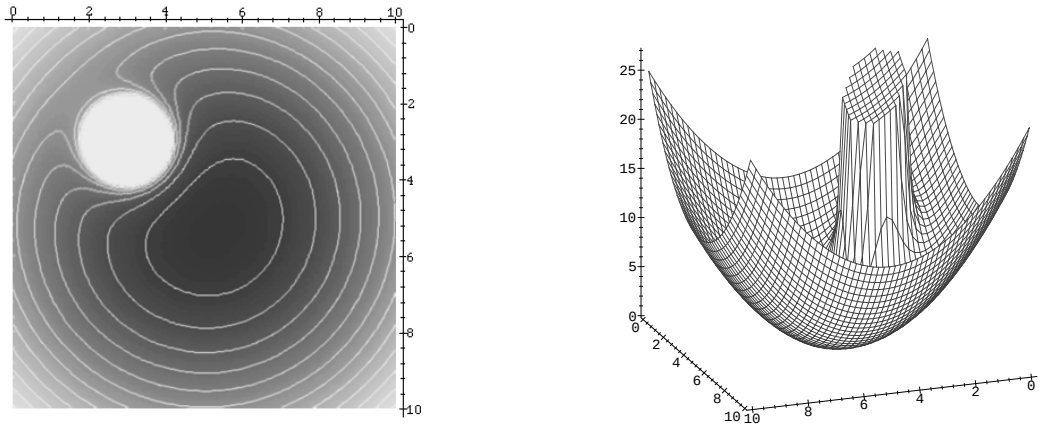


Figure 5.12: The repulsive and attractive potential fields.

5.9.1 Nowhere to Hide

The first animations we produced were to verify that the shark could easily catch a merperson swimming in open water. The shark is larger and swims faster, so it has no trouble catching its prey.

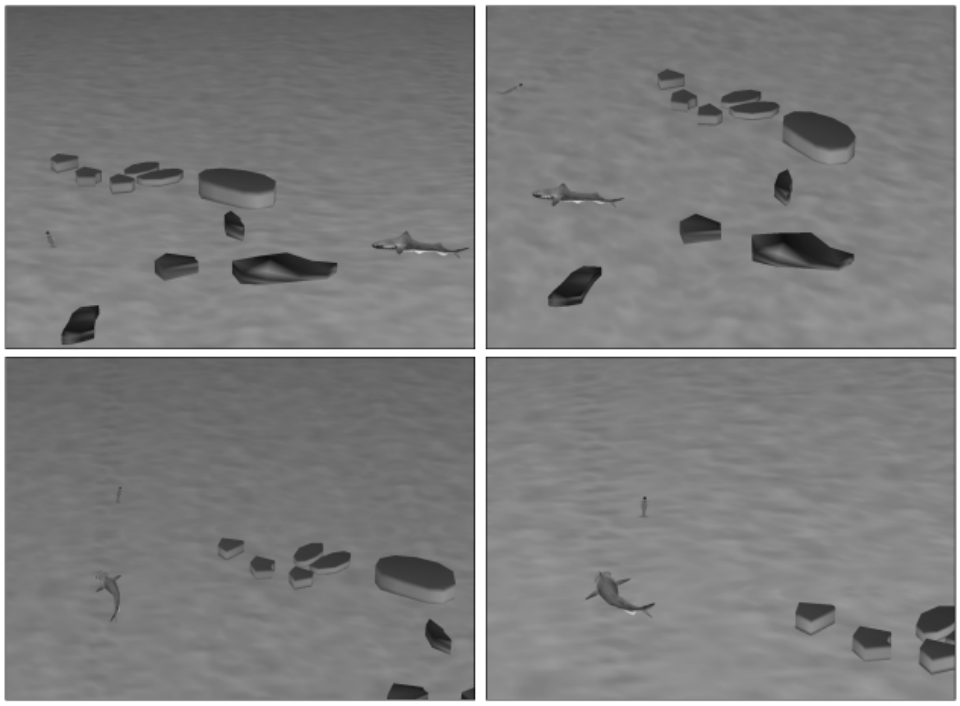


Figure 5.13: Nowhere to Hide (part I)

In particular, there are no obstacles/rocks that are large enough for the merman to hide behind. Therefore, as shown in figure 5.13, the merman simply tries to swim, as far, and as fast, as he can, away from the ferocious shark.

When he senses that the predator became dangerously close all he can do is go into panic mode, taking random turns... Unfortunately, the shark is larger and swims faster, it has no difficulty devouring its prey (see figure 5.14).

5.9.2 The Great Escape

In this animation, we use the same initial configuration as the previous one but introduce some large rocks into the merman's world. The merman takes advantage of undersea rocks to try and avoid being eaten. It can hide behind them and hug them closely so that the shark has difficulty seeing or reaching it. To cope with fast moving environments, the merpeople base their decisions on where to go on the positions that it predicts the predators will be in when it gets to its goal. So long as it was safe to do so, the merman will try to visit other obstacles. Finally, we enabled it to use information about relative character sizes to look for small gaps to go through whenever it had the chance.

Figure 5.15 shows some frames from the initial sequence. An intense chase ensues with the shark closely tailing its prey as it circles a big rock. If he thinks it is safe enough he will try to visit other obstacles in the scene, otherwise he will rest in his current hiding place for as long as he can. The initial attempts to try and visit other obstacles are thwarted as the shark swings back quickly and became too threatening again. Finally, the merman makes a break for the huge long rock towards the back of the scene and the shark pursues.

In figure 5.16 the shark starts gaining on the merman as the pair travel along the side of rock... Suddenly,

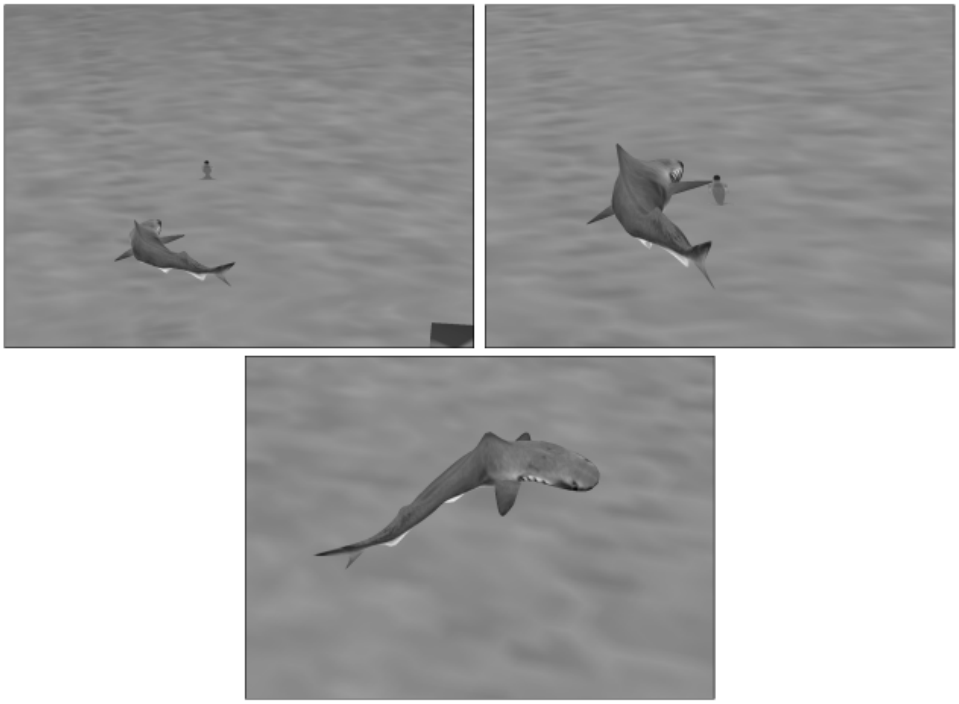


Figure 5.14: Nowhere to Hide (part II)

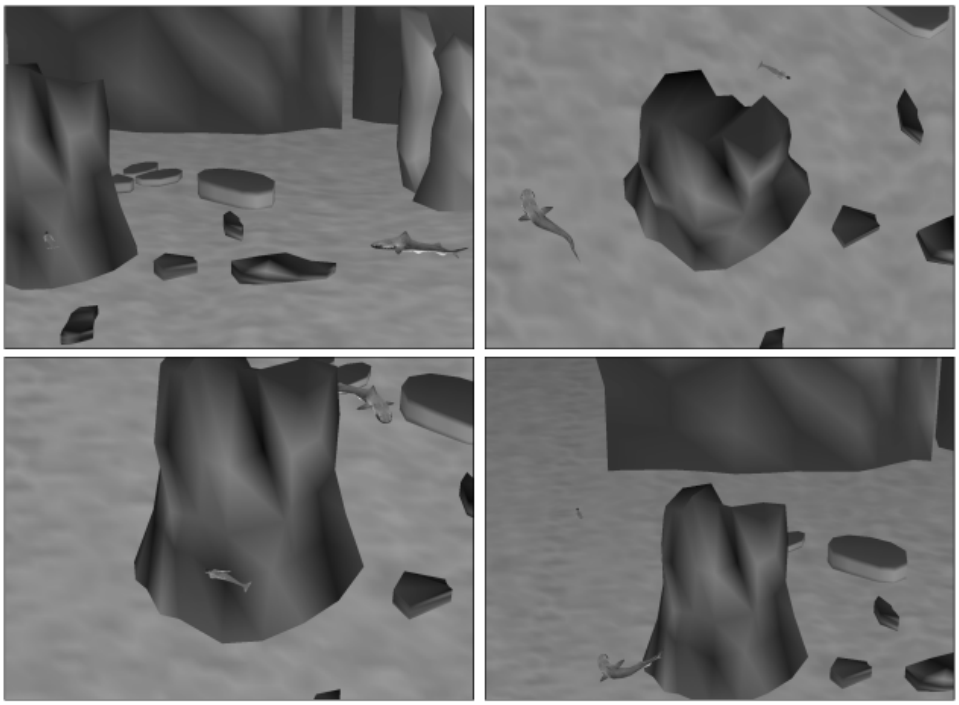


Figure 5.15: The Great Escape (part I)

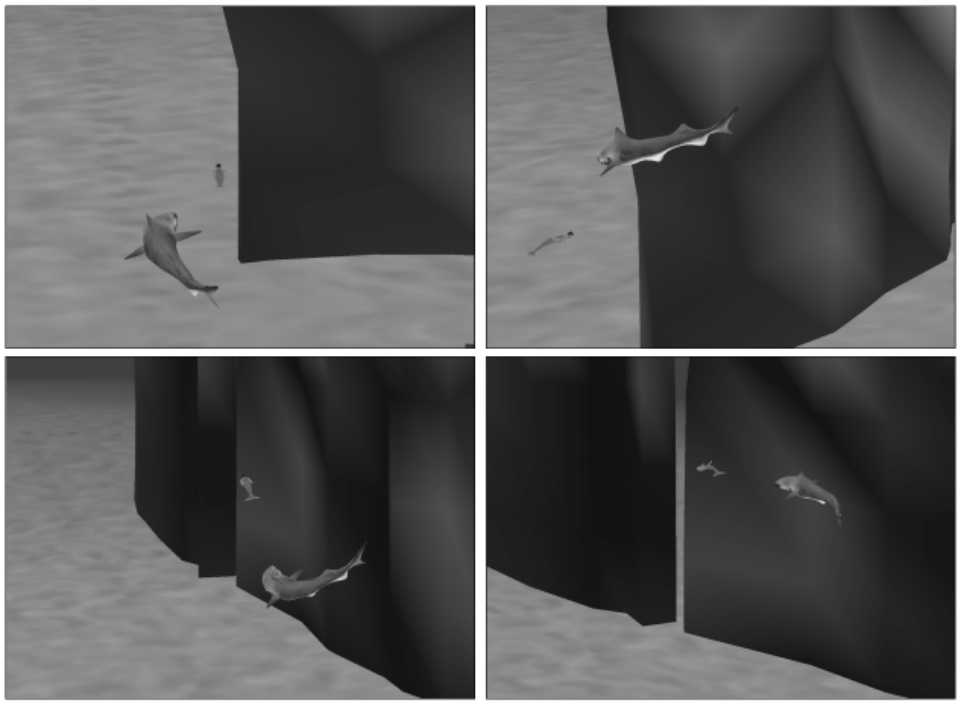


Figure 5.16: The Great Escape (part II)

the merman sees a small crack in the rock!

In figure 5.17 the merman swims through the crack. The shark tries to follow but luckily for the merman, the crack is too narrow for it to pass through without risking injury. The merman seizes the opportunity and the shark is foiled.

In general, the shark chases the merman if it can see him, otherwise it goes to check where it last saw him. We can see the shark following this behavior pattern as in the animation it returns to the crack in the rock.

Finally, in figure 5.18 the shark gives up and begins to search for the merman. By the time the shark reaches the other side of the rock, however, the merman is nowhere to be found.

5.9.3 Pet Protection

This simple animation shows that characters can have distinct "mersonalities" and can co-operate with each other. We have specified that some characters are brave and others are more timid. The timid ones cry for help (telepathically for now) when they are in danger and the brave ones will go to the rescue, provided it is not too dangerous for them. Once the brave ones have attracted the sharks attention, they try to get away themselves.

The small lighter colored creature in figure 5.20 is the merman's pet. It is timid and must be protected by the braver, and larger merperson. In the opening sequence the pet is being chased by a shark and calls out (telepathically for now) for help. The merman goes to its rescue. So long as the prey are in a certain range, the shark prefers larger meals. Hence it gave up on the small pet and started chasing the merman.

Figure 5.20 shows what happens now that the merman has got the shark's attention. The merman tries to escape and quickly hides behind a rock. The shark immediately discontinues the pursuit and wanders off back to where it last saw the merman's pet. Once again the merman charges to rescue and the shark again starts to pursue.

However this time he is too foolhardy and, as we can see in figure 5.21, is only saved from a grisly end

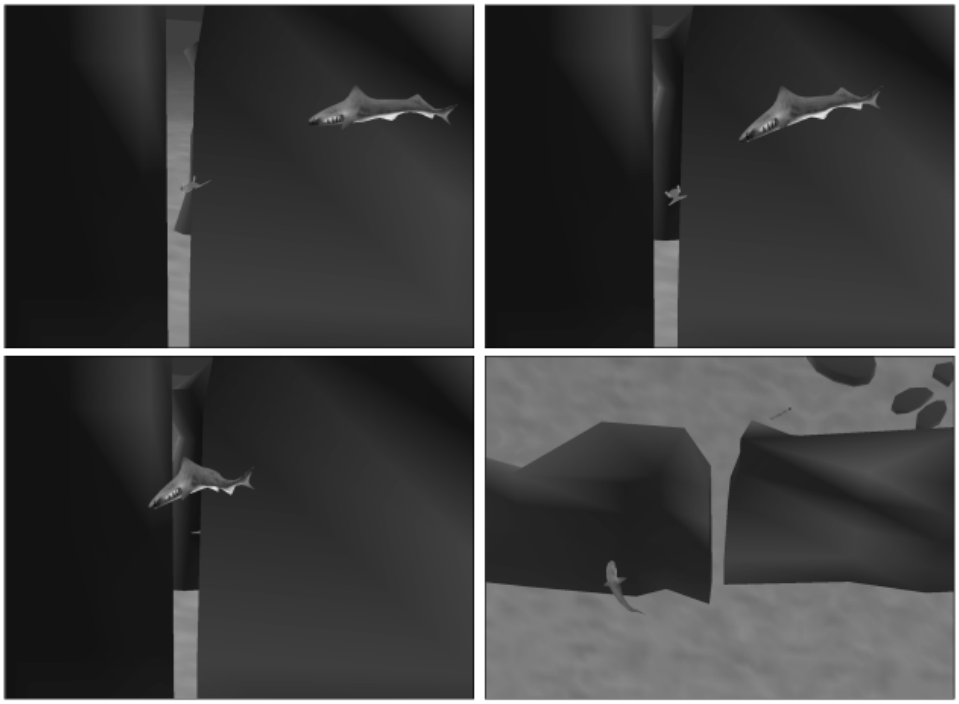


Figure 5.17: The Great Escape (part III)

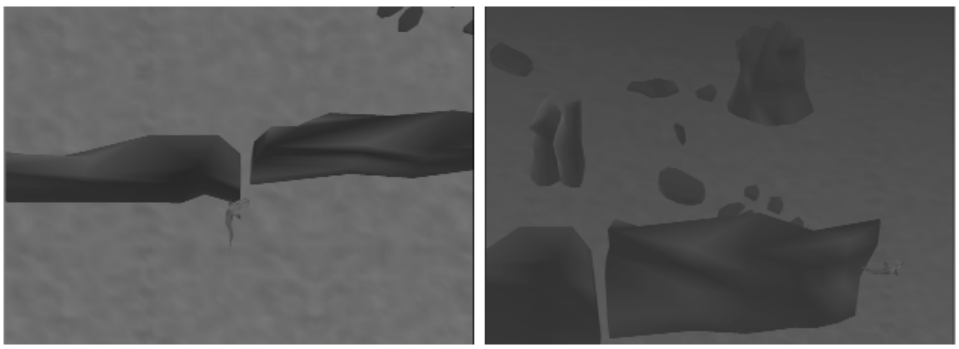


Figure 5.18: The Great Escape (part IV)

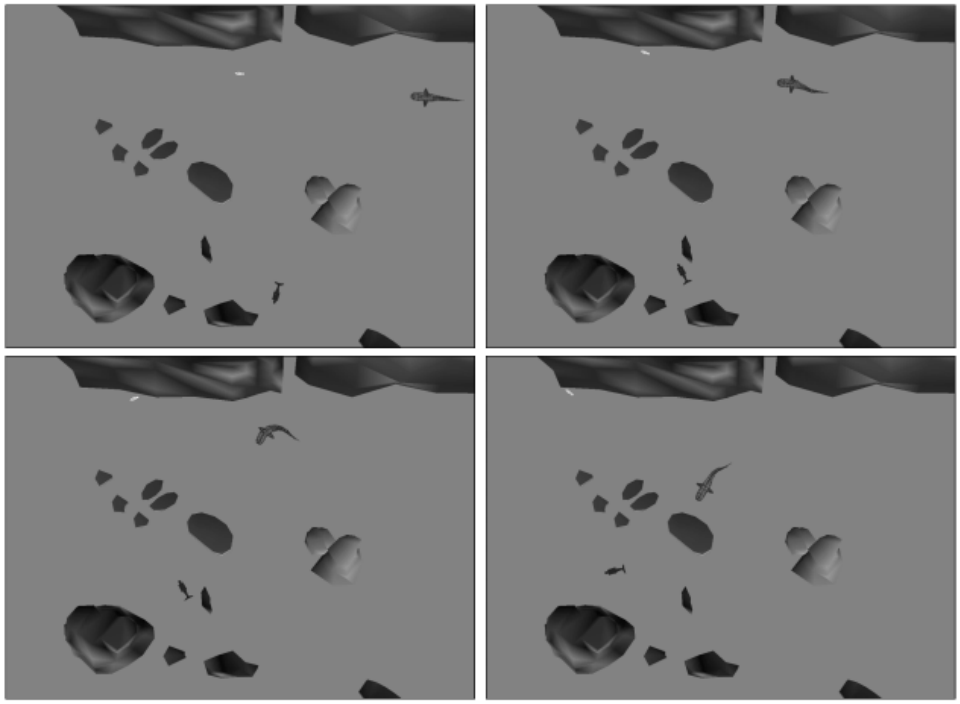


Figure 5.19: Pet Protection (part I)

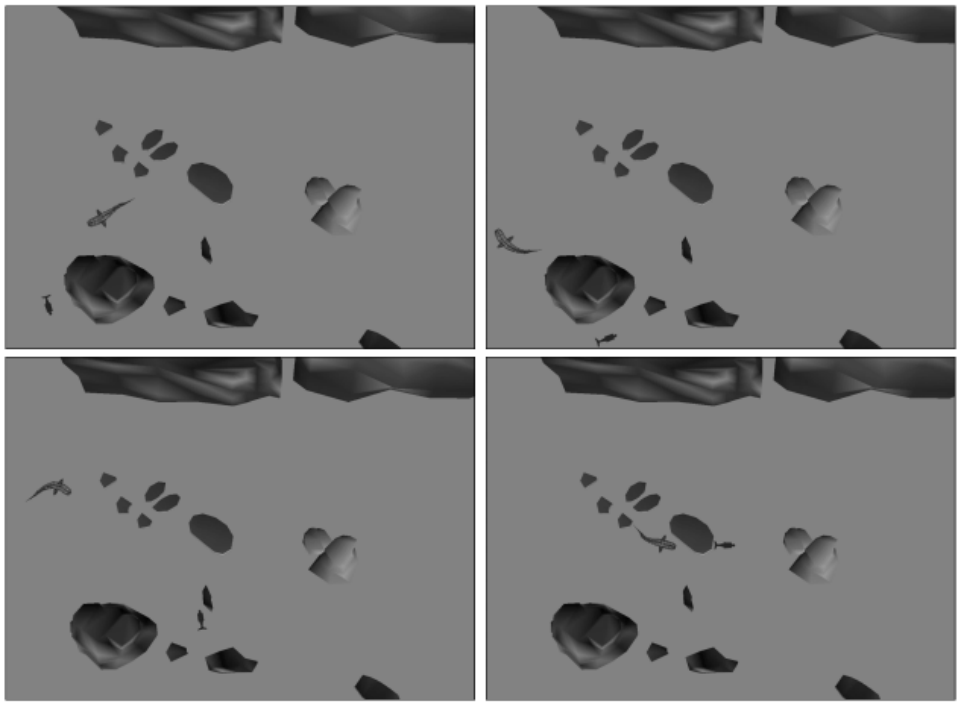


Figure 5.20: Pet Protection (part II)

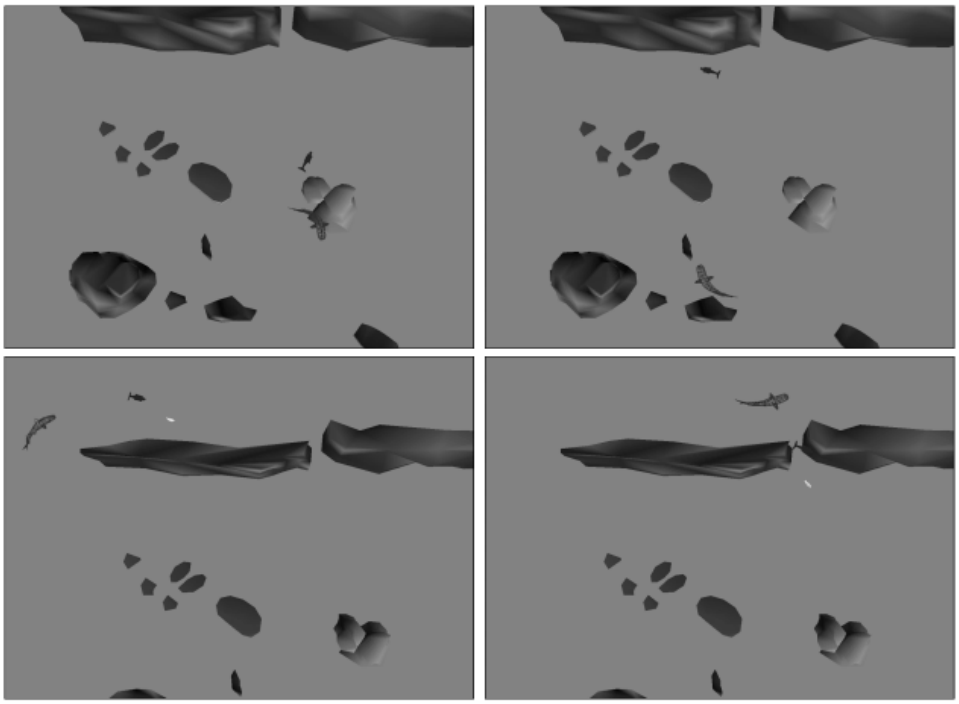


Figure 5.21: Pet Protection (part III)

by a lucky near miss that the shark has with an obstacle. Now the merman runs away from the shark and towards the last known position of his pet. The two of them then swim together and, as in the previous animation, evade capture.

5.9.4 General Mêlée

To show that our approach can scale, we have generated animations with larger numbers of characters. Each character has its own "brain", so there is no theoretical problem with adding as many as we want.

In figure 5.22 there are six merpeople and four sharks. The merpeople weigh the threat from each visible predator and at accordingly. This time they are all fending for themselves - although they are aware of all the other characters within visual range. The merpeople nearest the sharks start running first as they are the first to see them.

The action continues in figure 5.23 with the merpeople take advantage of their smaller size by hugging obstacles closely so that the sharks have difficulty reaching them. The birds-eye view also makes it hard to discern merpeople maneuvering over or under sharks from sharks just being merciful. Still one can see how for the most part the merpeople manage to stay clear of the sharks.

From a practical point of view, things start to get slow when we have over twenty reasoning characters in the scene. Improving the efficiency of our reasoning engine may help. We also want to consider using faster, limited models of reasoning. That is, we may obtain useful characters by removing tha ability to reason about disjunctions, say.

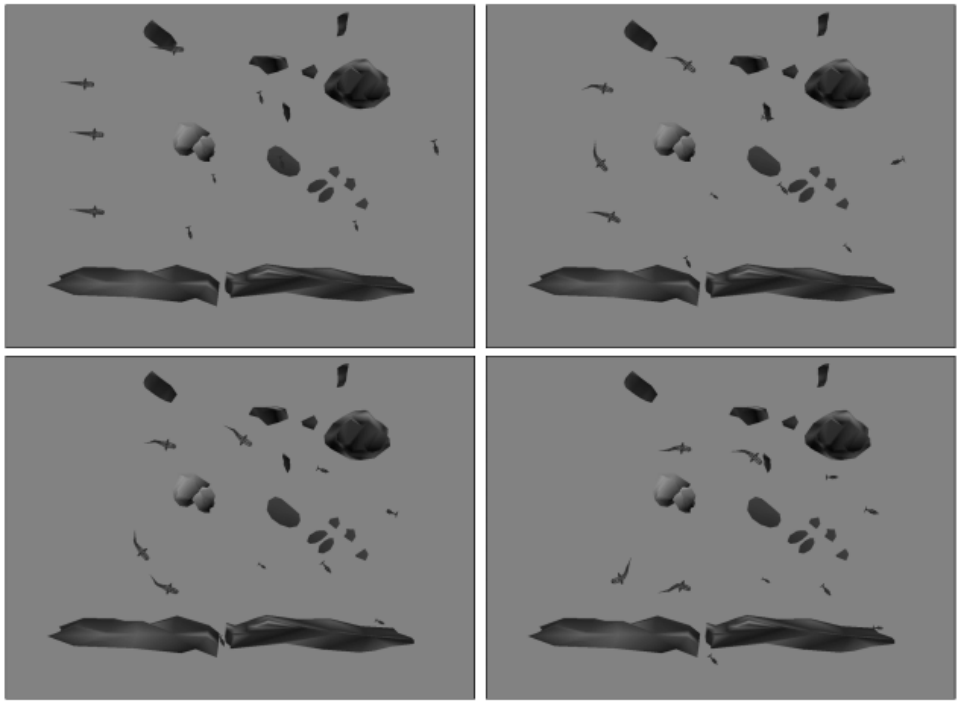


Figure 5.22: General Mêlée (part I)

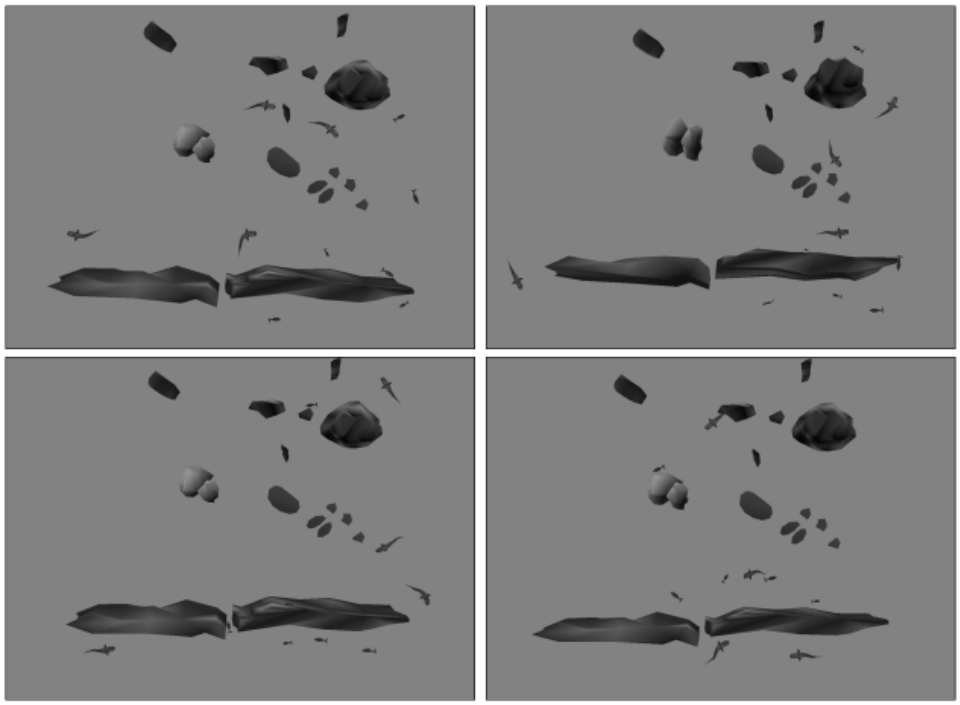


Figure 5.23: General Mêlée (part II)

Chapter 6

Conclusion

In this, our final chapter, we shall summarize the previous chapters, and point out the opportunities we perceive for extending our work.

6.1 Summary

The reader will recall that, after the introduction, the document began with a broad overview of computer animation. The chapter set the scene for what was to follow and gave the reader the required background to appreciate what was to follow.

Chapter 3 outlined the underlying theoretical basis for our approach. The most notable contribution was the introduction of interval-valued fluents. One of their key benefits was that they allowed us to elegantly talk about knowledge of continuous quantities.

We moved on to discuss applications of our approach to computer animation. We drew a distinction between the problems of controlling the behavior of a character situated in a physics based world, and a non-physics based world. In particular we first discussed the simpler case of a non-physics based world. We referred to these applications as kinematic. Although not generally the case, in our example the character in its kinematic world did not have to face any problems associated with their actions being unpredictable. We first discussed some simple examples that served as a concrete realization of how we meant to apply our theory to behavior specification. After which, we moved on to a real-world application of our theory to camera placement. We were able to demonstrate how well suited our notation was to such an endeavor.

The next step was to consider the case in which the character was situated in a hard to predict physics-based world. Sensing was the key to the successful application of our approach. We choose an undersea world as our scenario. A merman was the vehicle used to embody our ideas. The merpeople were able to use the behavior specifications we gave them to perform various tasks. We focussed on pursuit and evasion behaviors and generated some intriguing animations of merpeople and sharks chasing and avoiding one another. We explained that our system had numerous advantages in terms of rapid prototyping and maintainability.

6.2 Future Work

In order to make it easier for the reader to understand when best to use our approach it is important to discuss some of the current limitations. Many of these shortcomings are far from fundamental and thus provide a springboard for a discussion on possible future work.

- As it stands our approach is too slow to be used for interactive applications. We believe that in many interactive applications suitable allocation of CPU resources could allow computer characters to reason as a background process. The reasoning process would be given higher priority during natural pauses in the action. Some of the key factors to make this work are already in place. Firstly, our leverage

of the synergy between a reasoning system and a reactive system would be important in developing the "anytime algorithms" (see [129]) that are crucial for real-time applications. In addition, our use of intervals to represent partial knowledge of aspects of the world allows a character's knowledge to degrade gracefully over time.

- There is something to be said for being able to tell a character something before expecting it to learn it. However, learning is an important aspect of behavior that we have all but ignored. There are a number of ways we could redress the situation. Firstly, let us point out that there is little hope, in the short term at least, of learning high-level controllers. Such an endeavor would amount to the unrealistic goal of automatic programming. We can, however, imagine a character that can learn action precondition and effect axioms. It could do this by a process of experimenting with various actions that it is able to perform. Perhaps an even more exciting idea is to imagine a character that can observe the actions it decides upon by reasoning. Then, depending on what sensory values were used to make the decision, and how much planning was involved, the character could try and learn to mimic the working of the reasoning system. This would result in a function that, hopefully, maps sensory inputs, and internal state into appropriate actions. This function could be used in place of the reasoning system whenever time was limited. One could even imagine a learning phase during which a character wanders around its world trying things out, followed by a phase when it puts its newfound knowledge into action.

- One may think of our system as a providing a means to build arbitrary architectures for planning and control. To date, we have however only built simple action selection mechanisms. In contrast there are some highly sophisticated planning architectures such as SOAR [85], and RAP [20]. These systems make no pretense at being general purpose knowledge representation tools like ours. They are however extremely effective planners. To compete with them in this area we would ideally need to add additional features to our underlying representation language. In particular, back in chapter 5, section 5.8.6 we mentioned ConGolog [43]. ConGolog has many important features, such as parallelism and interrupts, that we would need to elegantly handle multiple goals and real-time constraints.

- A key problem with logical approaches to control is that once an inconsistency arises the whole system comes crashing to a halt. Currently the only recourse is to try and be careful and anticipate all eventualities. If instead it were possible to somehow introduce a localized notion of consistency then this would be of enormous practical benefit in terms of stability and robustness.

- Logic has many advantages as a precise means of communication. It would, however, be a worthwhile, and simple, enhancement to allow for an interaction language more akin to natural language. This would provide a (possibly pared down) version more suited to use by non-technical people. We might even consider a visual programming approach to specifying complex actions.

- In terms of implementation our choice of Prolog as a theorem prover is somewhat questionable. Prolog has some important differences with logic and thus dilutes our claims of mathematical rigor in specifying behaviors. Other theorem provers exist but many are heavily tailored toward interactively discovering proofs as opposed to application development. There are, however, some other candidates that we should evaluate [105].

- On the theoretical front we would like to try a more radical overhaul of the situation calculus. We have thus far introduced interval-valued fluents. We would like to try and move to a completely continuous model. That is, we would like a continuous notion of change, with continuous actions, continuous knowledge, continuous goals, etc. It is our belief that through the use of intervals we may effect such a scheme. One of the most significant technical impediments we envisage would be how to deal with overlapping actions and events. A key advantage, however, should be a practical theory that allows for graceful degradation as we reduce the amount of reasoning time available.

  Even if we stop short of such a grandiose achievement, we do not as yet take advantage of the full scope of the current available agent theory from which our approach derives [50]. Some of this theory is now fairly stable and perhaps amenable to incorporation into our work. Regardless of any unforeseen problems, our implementation domain provides an excellent opportunity for testing and, if necessary, refinement.

- The scope for applications of our work is obviously large. Our camera example obviously has plenty of room for expansion. We would also like to look at applications to lighting and sound production within virtual worlds.

  Other issues arise with regard to camera placement. It is common practice in computer animation to create a simulation and then to choose suitable camera placements. In film production the process is not so conveniently partitioned. It is commonplace to re-shoot a scene, with the actors in different positions (or even entirely removed from the scene!), and pretend that it is the same scene shot from a different angle. The reason for this approach are practical and aesthetic. In computer animation we do have the option of re-shooting the same piece of action from multiple viewpoints. That is, there are no practical reasons to re-enact a scene, but, it may well be the case that for aesthetic reasons the action needs to be changed depending on the viewpoint. For example, one might imagine a "face the camera if possible" behavior. This is an issue which has not been addressed within computer animation research.

  Other applications are also possible. Our work represents an excellent basis for continued research into high-level behavior control of autonomous animated characters. That is, our system can be used with any behavioral animation system for which the low-level control has been worked out. We would, therefore, like to apply our approach to the animation of some other creatures. The first obvious extension we wish to add to our system is to build a mermaid companion for our merman. Another particularly compelling example would be to incorporate our work into the process of computer game development. That is our approach could be used for rapid prototyping of new characters.

- The incorporation of more support for multiple characters would also be a lofty and worthy goal. We should like to implement some sophisticated level of detail scheme. We envisage that behavior could become simplified with distance from the camera, and flock membership.

- One aspect of the use of the situation calculus that we could exploit further is the ability to prove properties of our specifications. It seems that the computer industry remains resilient to acknowledging the advantages of proving properties of programs versus testing them. Regardless, it is an idea whose time will no doubt come. Our work will be uniquely poised to capitalize on any such a paradigm shift.

- Finally, we believe the widespread protection of computer characters as intellectual property will one day become a reality. The problems associated with legal wrangling over copyright infringement could be easily dispelled by recourse to a formalism such as ours. The advantage to this is, of course, that we would have the opportunity to prove or disprove the alleged similarities.

6.3 Conclusion

In advanced computer animation research, we study virtual autonomous agents situated in dynamic virtual worlds. In the past, researchers in computer animation have used techniques from control theory and numerical optimization to allow them to address the low-level locomotion control problem. We have applied a theory of action that allows animated autonomous characters to perceive, reason and act in dynamic virtual worlds. We have proposed and implemented a remarkably useful framework for high-level control that combines the advantages of a reactive and a reasoning system. It allows us to conveniently specify the high-level behavior we want our animated characters to exhibit. We have used this system to create sophisticated behavioral animations with unprecedented ease.

In summary, we have developed a behavioral animation framework that supports convenient high-level interaction and direction of autonomous virtual agents by:

1. Enabling an agent to represent and think about its worlds in a way that is intuitive to the animator;
2. Allowing a user to define the agents representation of its world in terms of actions and their effects;
3. Combining the advantages of a high-level reasoning system with a lower level reactive behavior system;
4. Enabling a reasoning agent to sense, so that it can be situated in a physics-based dynamic world;

5. Enabling an agent to receive advice on how to behave in the form of a sketch plan, the details of which the agent can automatically infer;

6. Being able to direct an agent without sacrificing its autonomy at run-time.

Much of the early work in computer animation consisted of producing efficient implementations of perfectly good scientific theories, or failing that, approximations to existing theories that still produce realistic looking results. It was not long before difficult unsolved problems, such as the control problem for physics based animations, presented themselves. For many aspects of the world, most notably cognitive models (including control), scientific theory is much less complete and computer animation should play an integral role, both as a test bed and a driving force, in developing new ideas.

Appendix A

Scenes

A *scene* shall be taken to be the state of some world at some particular time. To talk about the geometrical component of a scene the following, standard, definitions are introduced:

Points. A point is an atomic location in space. The space as a whole is a set of points.

Lengths. Length is a differential measure space. Note that this definition also gives rise to areas (length squared) and volume (length cubed).

Directions. A direction is a point on the unit sphere centered at the origin.

Coordinate Systems. A right-handed coordinate system in n-space is a triple $\mathcal{C} = (O, l, \mathbf{D})$, where O is a point called the origin, l is a unit length, and $\mathbf{D}$ is the frame of axis directions (or coordinate frame), an n-tuple of mutually perpendicular directions, ordered with the "right-hand" orientation.

Fixing a coordinate system gives a standard way of naming points, lengths and directions:

Let $\mathcal{C} = (\mathbf{o}, l, (\mathbf{d}_0, \ldots, \mathbf{d}_{n-1}))$ be a coordinate system in n-space. The measure of length m in $\mathcal{C}$ is the real number m/l. The product $l\mathbf{d}$ is the vector with length l and direction $\mathbf{d}$. The coordinates of a vector $\mathbf{v}$ is the unique n-tuple $c_0, \ldots c_{n-1}$ such that $v = c_0 l\mathbf{d}_0 + \cdots + c_{n-1} l\mathbf{d}_{n-1}$. The coordinates of a point $\mathbf{a}$ in $\mathcal{C}$ is equal to the coordinates of the vector $\mathbf{a} - \mathbf{o}$.

Mappings. A mapping is a function from the space to itself.

Regions. A region is a set of points.

Although there has been some interest in two-dimensional computer animation, most of the work considered in this paper deals with, or is applicable to, three-dimensional computer animation. In particular the following three-dimensional world coordinate system is defined $\mathcal{C}_0 = (\mathbf{O}, 1, (\mathbf{i}, \mathbf{j}, \mathbf{k}))$, where $\mathbf{O} = (0, 0, 0)$, $\mathbf{i} = (1, 0, 0)$, $\mathbf{j} = (0, 1, 0)$ and $\mathbf{k} = (0, 0, 1)$.

Appendix B

Homogeneous Transformations

Let $\mathbf{q} = (q_1, q_2, q_3)$ be a point in $\mathbb{R}^3$, $\mathcal{C}$ be a coordinate system for $\mathbb{R}^3$ then the homogeneous coordinates of $\mathbf{q}$ with respect to $\mathcal{C}$ are defined as:

$$[\mathbf{q}]^{\mathcal{C}} = (\sigma q_1, \sigma q_2, \sigma q_3, \sigma)^T$$

For convenience, in animation and robotics, it is usual to take $\sigma = 1$. Thus, four-dimensional homogeneous coordinates can be obtained from three-dimensional physical coordinates by simply augmenting the vector with a unit fourth component. Similarly, three-dimensional physical coordinates are recovered from four-dimensional homogeneous coordinates by merely dropping the unit forth component. Where the intended coordinate system and use of homogeneous coordinates is not ambiguous $\mathbf{q}$ will be written instead of $[\mathbf{q}]^{\mathcal{C}}$.

A homogeneous transformation of points in $\mathbb{R}^3$ can be represented by a 4×4 matrix $\mathbf{T}$ of the form:

$$\mathbf{T} = \begin{pmatrix} \mathbf{R} & \mathbf{p} \\ \boldsymbol{\eta}^T & \sigma \end{pmatrix}$$

Where $\mathbf{R}$ is a compound 3×3 shear and rotation matrix, $\mathbf{p}$ is a translation vector, $\boldsymbol{\eta}$ is a perspective vector and σ is a scalar scale factor. By setting $\boldsymbol{\eta} = \mathbf{0}$, $\sigma = 1$ and restricting $\mathbf{R}$ to be purely a rotation matrix it is possible to represent a rigid body motion.

Appendix C

Control Theory

A more comprehensive introduction to control theory may be found in any good control theory book, such as [36]. Most of our definitions are taken from [33].

A *process* is a series of changes in the state of some world. A process may also be referred to as a *plant*, or the *environment*. A *controller* changes the state of the world to influence what additional changes will occur and when.

The mathematical model of the interaction between the environment and the controller is an example of a *dynamical system*. The set of all times is denoted $\mathfrak{T}$ and may be continuous or discrete but for any computer animation problem will always be bounded. The set of all possible states of the environment is known as the *state space* $\mathfrak{X}$ of the dynamical system. The space of all possible *trajectories* (or *time lines*) is defined as the set of all functions from time to states: $H_{\mathfrak{X}} = \{\mathbf{x} : \mathfrak{T} \to \mathfrak{X}\}$. A point in state space $\mathbf{x}(t)$ is known as a *state vector*, it completely describes the state of the environment at any given time $t \in \mathfrak{T}$. The set of *outputs* $\mathfrak{Y}$ denotes the set of all the things the controller can perceive of its environment. The space of all output trajectories is defined as the set of all functions from time to outputs: $H_{\mathfrak{Y}} = \{\mathbf{y} : \mathfrak{T} \to \mathfrak{Y}\}$. A point in output space is denoted by the vector $\mathbf{y}(t)$. The set of *inputs* $\mathfrak{V}$ denotes the set of all possible *actions* that the controller can perform. The space of all input trajectories is defined as the set of all functions from time to inputs: $H_{\mathfrak{V}} = \{\mathbf{v} : \mathfrak{T} \to \mathfrak{V}\}$. A point in input space is denoted by the vector $\mathbf{v}(t)$.

The state trajectories are generally restricted to obey certain laws, as embodied in the *system state equations*[1]: $\dot{\mathbf{x}} = \mathbf{k}(\mathbf{x}(t), \mathbf{v}(t))$.

The output function $\mathbf{y}(t)$ restricts the set of output trajectories by relating them to states: $\mathbf{y}(t) = \mathbf{g}(\mathbf{x}(t))$. Different types of output function can arise:

Open-loop. In an open-loop controller the output function is just the identity function: $\mathbf{y}(t) = t$. An open-loop controller must therefore be recalculated for each new initial situation.

Closed-loop. In a closed-loop controller the output function returns information about the current state. This information is known as *feedback*. Feedback makes the controller applicable to a range of initial situations.

Among the different types of feedback are:

State Feedback. State feedback implies that the function $\mathbf{g}$ is just the identity function so that $\mathbf{y}(t) = \mathbf{x}(t)$. It thus makes complete information about the state of the system available to the controller;

Sensor Feedback. Sensor feedback implies that the function $\mathbf{g}$ is not just the identity function. It may or may not be possible to reconstruct a complete state description from the information provided.

[1] Here the system state equation is a system of differential equations, however it might just as well be a difference equation, a set of axioms or a stochastic process.

The set of input trajectories are restricted by the resources available (or in computer animation, that would realistically be available) to build a controller. The set of all possible controllers is the set of all functions, known as *control functions* (or *control laws*), from output trajectories to inputs: $P = \{\mathbf{p} : H_{\mathfrak{Y}} \rightarrow \mathfrak{V}\}$. Often only the last state is considered relevant, in which case: $P = \{\mathbf{p} : \mathfrak{Y} \rightarrow \mathfrak{V}\}$.

Specifying what a controller should do can be done in various ways:

Hand-crafting. *Hand-crafting* a controller consists of directly defining a control function.

Optimal control. In the *optimal control* approach a scalar function $o : H_{\mathfrak{X}} \rightarrow \mathbb{R}$, called a *performance index* (or *objective function*), is used to induce a *total order* on the space of state space trajectories. Here the problem is to find an *optimal controller* $\mathbf{p} \in P$ such that (without loss of generality) o obtains a minimum value. It is implicitly assumed that the controller can be generated using optimization.

Objective based control. An objective based controller is one in which what the controller should do is specified, in terms of preferred states, as an *objective*. For example a *goal* $G \subset H_{\mathfrak{X}}$ is a set of possible state trajectories, so here the objective is to find a $\mathbf{p} \in P$ such that the behavior of the dynamical system is restricted to G. The point is that no means is given to achieve the goal.

It is also common to draw distinctions based on the types of systems being controlled. For example, a system described by a set of equations such as the following, is an instance of a linear system:

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{v} \\ \mathbf{y} &= \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{v} \end{aligned}$$

where $\mathbf{A}, \mathbf{B}, \mathbf{C}$ and $\mathbf{D}$ are all real-valued constant matrices. Linear systems are relatively easy to control and an extensive body of literature exists on the subject. In particular there are many efficient techniques for finding an optimal controller for a linear system.

Other classifications based on the type of system being controlled include:

Smooth Systems. A system is smooth if its state variables are C^1 continuous with respect to time. This is an important sub-category because it means that the derivates are available to optimization routines employed in the calculation of some optimal control strategy.

Statically Stable Systems. In computer animation people often refer to statically stable systems. By this they mean one in which momentum is not required for stability. Thus a motion in a statically stable system may be stopped at any point without the system collapsing.

A large amount of control theory is concerned with proving stability and performance under all possible conditions. This has thus far not been considered relevant to animation and is therefore not discussed further.

Appendix D

Complex Actions

Complex actions are abbreviations, or macros, for terms in the situation calculus. The notion is straightforward and similar to work done in proving properties of computer programs [46]. Our definitions are taken from those given in [68].

Complex actions are represented by the macro $\mathit{Do}(\alpha, s, s')$, such that s' is a state that results from doing the complex action α in state s. Do is defined recursively as follows:

$$
\begin{aligned}
\mathit{Do}(\alpha, s, s') &\triangleq \mathit{Poss}(\alpha, s) \wedge s' = \mathit{do}(\alpha, s) \quad \alpha \text{ is a primitive action} \\
\mathit{Do}((\alpha \fatsemi \beta), s, s') &\triangleq \exists s^* \ (\mathit{Do}(\alpha, s, s^*) \wedge \mathit{Do}(\beta, s^*, s')), \quad \text{sequence;} \\
\mathit{Do}(\phi?, s, s') &\triangleq \phi[s] \wedge s = s', \quad \text{test actions,} \\
&\quad \text{(where } \phi[s] \text{ is } \phi \text{ with situation arguments inserted,} \\
&\quad \text{eg. if } \phi = \mathsf{PreyPos}(p) \text{ then } \phi[s] = \mathsf{PreyPos}(p, s)); \\
\textbf{if } p \textbf{ then } \alpha \textbf{ else } \beta &\triangleq (p? \fatsemi \alpha) | (\neg p? \fatsemi \beta), \quad \text{conditionals;} \\
\mathit{Do}(\alpha^\star, s, s') &\triangleq \forall P \ \{[\forall s_1 \ P(s_1, s_1)] \wedge \\
&\qquad \forall s_1, s_2, s_3 \ [P(s_1, s_2) \wedge \mathit{Do}(\alpha, s_2, s_3) \Rightarrow P(s_1, s_3)]\} \\
&\qquad \Rightarrow P(s, s'), \quad \text{nondeterministic iteration;} \\
\textbf{while } p \textbf{ do } \alpha \textbf{ od} &\triangleq (p? \fatsemi \alpha)^\star | (\neg p?), \quad \text{while loops;} \\
\mathit{Do}((\alpha | \beta), s, s') &\triangleq \mathit{Do}(\alpha, s, s') \vee \mathit{Do}(\beta, s, s'), \quad \text{nondeterministic action choice;} \\
\mathit{Do}((\pi x)\alpha(x), s, s') &\triangleq \exists x \ \mathit{Do}(\alpha(x), s, s'), \quad \text{nondeterministic choice of arguments;} \\
\mathit{Do}(P(x_1, \ldots x_n), s, s') &\triangleq P(x_1[s], \ldots x_n[s], s, s') \\
\mathit{Do}((\textbf{proc } P(x_1, \ldots, x_n)\alpha \textbf{ end}) \ \beta, s, s') & \\
&\triangleq \forall P \ [\forall x_1, \ldots, x_n, s_1, s_2 \ \mathit{Do}(\alpha, s_1, s_2) \Rightarrow \\
&\quad \mathit{Do}(P(x_1, \ldots, x_n), s_1, s_2)] \Rightarrow \mathit{Do}(\beta, s, s'), \\
&\qquad \text{situation calculus version of standard} \\
&\qquad \text{Scott-Strackey [108] least fixed-point definition of} \\
&\qquad \text{(recursive) procedure execution.}
\end{aligned}
$$

Appendix E

Implementation

In [68], an implementation of the situation calculus is described. The implementation is known as Golog and provides a novel high-level programming language for controlling autonomous agents. In essence, the user supplies successor state axioms, precondition axioms and complex actions. The axioms are written directly in Prolog. This makes sense because the connection with logic programming means that there should be Prolog constructs that are closely related to the logical constructs used in the situation calculus. The complex actions are written in Golog and there is an interpreter to perform the macro expansion described in appendix D. The interpreter is written in Prolog. This seems to make less sense.

To be precise about where our objections lie let us clearly state that, for the same reasons as given above, it seems entirely reasonable to have the complex actions expand out into Prolog terms. What is not clear is why the macro expansion itself should be defined in Prolog. In particular, the macro expansion is an extra-logical notion so a tie-in with logic programming has dubious value. It would seem that the only tangible outcome of performing the macro expansion in Prolog, at run-time is to make the execution time unnecessarily slow.

Therefore, we have written what amounts to a Golog compiler. The compiler takes as input valid Golog programs and produces, as output, equivalent Prolog programs. Thus, at run-time, there is no need to interpret the Golog code on the fly. That is, the conversion to Prolog is done as a pre-processing step.

Our current compiler is a prototype version written in Java using the Java Compiler Compiler (JavaCC) [78]. Aside from the potential efficiency advantages, our compiler allows much better looking programs to be written. It does not require the use of cryptic and unseemly brackets. We can have arbitrary blank spaces, blank lines, tabs and it supports single and multi line comments. Note also that we are not forced to use "`:`" as the statement separator, but can use the more usual "`;`". In Prolog "`;`" is reserved to mean "or" and thus causes confusion. In future we would like to take advantage of Java's ability to handle Unicode characters [116]. This would result in programs that are syntactically identical to the theoretical language.

Golog stand for "alGOL in LOGic". The syntax of our complex actions differs slightly from that used in Golog. In particular, we wanted to make the approach accessible to as wide an audience as possible. We therefore prefer to make our language resemble C more closely than Algol. To avoid confusion we shall therefore refer to our language as CLog.

Below we shall describe how our compiler works. We will do this by going through the different CLog constructs and writing down the corresponding Prolog code that our compiler produces. The compiler is also available as an applet on the web [59] so that the interested reader may try out the examples for themselves.

We shall take advantage of the following background domain knowledge. To ensure that this Prolog code is loaded in the code output from CLog the user simply needs to place it in a file called `simple.pl`, say and

use the CLog statement `import simple;`.

```
global_var(max,3).

notequal(X,Y) :-
  \+ X = Y.

poss(act_inc,S) :-
  fl_counter(X,S),
  global_var(max,Max),
  X < Max.

poss(act_set(_),_).

fl_counter(Count,do(A,S)) :-
  (
    A = act_inc,
    fl_counter(OldCount,S),
    Count is OldCount + 1
  ) ;
  (
    A = act_set(X),
    Count is X
  ) ;
  (
    notequal(A,act_inc),
    notequal(A,act_set(_)),
    fl_counter(Count,S)
  ).

fl_counter(0,s0).
```

This successor state axiom states that the action `act_inc()` increments the counter by one, and is possible provided the counter is less than some maximum value (3 in this case). The action `act_set(X)` is always possible and sets the counter to `X`. Initially the counter is 0.

E.1 Overall structure

CLog works by considering each statement in the CLog program in turn. The top level clause is called `clog(S)`. This is the name of the clause the user needs to call to run the CLog program after is has been compiled. The result of a call to `grun(S)` will be to bind the variable S to the resulting sequence of primitive actions.

Let us consider a simple example to explain how things work. The simplest program is the empty program, and the compiled version of the empty program is another empty program. The next simplest valid CLog program is:

```
;
```

For both the empty program, and the above program, the resulting Prolog program is:

```
clog(S) :-
  clog_0(s0,S).

clog_0(S,S).
```

Thus, the result of a call to `clog(S)` will be that `S` is bound to `S = s0`, as required.

Now consider the program that consists of a single primitive action:

```
import simple;

act_inc();
```

The resulting Prolog program is :

```
:- ensure_loaded(simple).

clog(S) :-
  clog_0(s0,S).

clog_0(S,SNew) :-
  poss(act_inc,S),
  clog_1(do(act_inc,S),SNew).

clog_1(S,S).
```

Therefore, provided the `act_inc` action is possible in `s0` the result of a call to `clog(S)` will be that `S` is bound to `S = do(act_inc,s0)`, as required. Thus the Prolog query `clog(S), fl_counter(X,S)`, results in `X = 1`, as required. If we set `global_var(max,0).`, then the action is not possible, and the program will fail.

Notice that for actions with no arguments CLog strips off the `()`. This is because Prolog generates an error on reading such a construct. Ideally we would prefer to leave them in.

As required for the nondeterministic constructs in CLog, failure of a corresponding clause will result in backtracking to try and find other solutions. The clauses generated by the compiler are all named `clog_n`, where `n` is a number that gets incremented as we consider more and more statements in the CLog program. JavaCC has a companion tool called JJTree that acts as a preprocessor for JavaCC. JJTree generates code to construct parse tree nodes for each nonterminal in the language. We exploit the tree structure to ensure that we number all the clauses correctly. How the numbering scheme works will become clearer when we consider more examples. For a long program we can generate hundreds of clauses. It is the job of the compiler to keep track of the clause numbers and to assign them correctly.

To make it easier to understand we shall consider short programs that contain only one construct at a time. Of course with suitable re-numbering of the clauses we can imagine that they are fragments of a larger program. So far we have considered primitive actions. We shall now proceed to consider other cases.

E.2 Sequences

Consider the CLog program

```
import simple;

act_set(0);
act_inc();
```

The resulting Prolog program is

```
:- ensure_loaded(simple).

clog(S) :-
  clog_0(s0,S).

clog_0(S,SNew) :-
  poss(act_set(0),S),
  clog_1(do(act_set(0),S),SNew).

clog_1(S,SNew) :-
  poss(act_inc,S),
  clog_2(do(act_inc,S),SNew).

clog_2(S,S).
```

The result of the Prolog query `clog(S)` is `S = do(act_inc,do(act_set(0),s0))`, as required.
Equally as important is the fact that the following program results in a Prolog program for which the query `clog(S)` correctly replies "no".

```
import simple;

act_inc();
act_inc();
act_inc();
act_inc();
```

E.3 Tests

Consider the CLog program

```
import simple;

fluent fl_counter;

test(fl_counter(X) && X < 1);
```

The resulting Prolog program is

```
:- ensure_loaded(simple).

clog(S) :-
  clog_0(s0,S).

clog_0(S,SNew) :-
  fl_counter(X,S),
  X<1,
  clog_1(S,SNew).

clog_1(S,S).
```

Notice that we need to declare `fl_counter` as a fluent so that CLog knows to insert the appropriate situation argument whenever it is mentioned in a program.

E.4 Conditionals

Consider the CLog program

```
import simple;

fluent fl_counter;

if (fl_counter(X) && X < 1)
   act_inc();
```

The resulting Prolog program is

```
:- ensure_loaded(simple).

clog(S) :-
  clog_0(s0,S).

clog_1(S,SNew) :-
  poss(act_inc,S),
  clog_2(do(act_inc,S),SNew).

clog_0(S,SNew) :-
  fl_counter(X,S),
  X<1,
  clog_1(S,SNew).

clog_0(S,SNew) :-
  \+ (fl_counter(X,S),
  X<1),
  clog_2(S,SNew).

clog_2(S,S).
```

The result of the Prolog query `clog(S)` is `S = do(act_inc,s0)`, as required. Now consider the same CLog program with an else part.

```
import simple;

fluent fl_counter;

if (fl_counter(X) && X < 1)
   act_inc();
else
   act_set(0);
```

The resulting Prolog program is

```
:- ensure_loaded(simple).

clog(S) :-
  clog_0(s0,S).

clog_1(S,SNew) :-
  poss(act_inc,S),
  clog_2(do(act_inc,S),SNew).

clog_3(S,SNew) :-
  poss(act_set(0),S),
  clog_4(do(act_set(0),S),SNew).
```

```
clog_0(S,SNew) :-
  fl_counter(X,S),
  X<1,
  clog_1(S,SNew).

clog_0(S,SNew) :-
  \+ (fl_counter(X,S),
  X<1),
  clog_3(S,SNew).

clog_2(S,SNew) :-
  clog_4(S,SNew).

clog_4(S,S).
```

The result of the Prolog query `clog(S)` is `S = do(act_inc,s0)`, as required. Of course, we can also have a block of statements wherever we could have a single statement. For example, consider the CLog program

```
import simple;

fluent fl_counter;

if (fl_counter(X) && X < 1) {
   act_set(0);
   act_inc();
}
```

The resulting Prolog program is

```
:- ensure_loaded(simple).

clog(S) :-
  clog_0(s0,S).

clog_1(S,SNew) :-
  poss(act_set(0),S),
  clog_2(do(act_set(0),S),SNew).

clog_2(S,SNew) :-
  poss(act_inc,S),
  clog_3(do(act_inc,S),SNew).

clog_0(S,SNew) :-
  fl_counter(X,S),
  X<1,
  clog_1(S,SNew).

clog_0(S,SNew) :-
  \+ (fl_counter(X,S),
  X<1),
  clog_3(S,SNew).

clog_3(S,S).
```

The result of the Prolog query `clog(S)` is `S = do(act_inc,do(act_set(0),s0))`, as required.

E.5 Nondeterministic iteration

Consider the CLog program

```
import simple;

star
  act_inc();
```

The resulting Prolog program is

```
:- ensure_loaded(simple).

clog(S) :-
  clog_0(s0,S).

clog_1(S,SNew) :-
  poss(act_inc,S),
  clog_2(do(act_inc,S),SNew).

clog_0(S,SNew) :-
  clog_3(S,SNew).

clog_0(S,SNew) :-
  clog_1(S,SNew).

clog_2(S,SNew) :-
  clog_0(S,SNew).

clog_3(S,S).
```

The result of the Prolog query `clog(S)` is `S = s0`, or `S = do(act_inc,s0)`, or `S = do(act_inc,do(act_inc,s0))`, or `S = do(act_inc,do(act_inc,do(act_inc,s0)))`, as required.

The next program will be of interest for the while construct that we consider next.

```
import simple;

fluent fl_counter;

star
  {
  test(fl_counter(X) && X < 2)
  act_inc();
  }
test(!(fl_counter(X) && X < 2));
```

The resulting Prolog program is

```
:- ensure_loaded(simple).

clog(S) :-
  clog_0(s0,S).

clog_1(S,SNew) :-
  fl_counter(X,S),
  X<2,
  clog_2(S,SNew).
```

```
clog_2(S,SNew) :-
  poss(act_inc,S),
  clog_3(do(act_inc,S),SNew).

clog_0(S,SNew) :-
  clog_4(S,SNew).

clog_0(S,SNew) :-
  clog_1(S,SNew).

clog_3(S,SNew) :-
  clog_0(S,SNew).

clog_4(S,SNew) :-
  \+ ((fl_counter(X,S),
  X<2)),
  clog_5(S,SNew).

clog_5(S,S).
```

The result of the Prolog query `clog(S)` is `S = do(act_inc,do(act_inc,s0))`, as required.

E.6 While loops

Consider the CLog program

```
import simple;

fluent fl_counter;

while ((fl_counter(X) && X < 2))
  act_inc();
```

Notice that from the definitions given in appendix D this is the same program as the previous one involving `star` and `test`. The resulting Prolog program is

```
:- ensure_loaded(simple).

clog(S) :-
  clog_0(s0,S).

clog_1(S,SNew) :-
  poss(act_inc,S),
  clog_2(do(act_inc,S),SNew).

clog_0(S,SNew) :-
  (fl_counter(X,S),
  X<2),
  clog_1(S,SNew).

clog_0(S,SNew) :-
  \+ ((fl_counter(X,S),
  X<2)),
  clog_3(S,SNew).
```

```
clog_2(S,SNew) :-
  clog_0(S,SNew).

clog_3(S,S).
```

Note the code generated is different to the previous example. In particular it is optimized for the while loop. As expected, it does however produce the same sequence of primitive actions. In particular, the result of the Prolog query `clog(S)` is `S = do(act_inc,do(act_inc,s0))`, as before.

E.7 Nondeterministic choice of action

Consider the CLog program

```
import simple;

choose
 act_inc();
or
 act_set(2);
```

The resulting Prolog program is

```
:- ensure_loaded(simple).

clog(S) :-
  clog_0(s0,S).

clog_1(S,SNew) :-
  poss(act_inc,S),
  clog_2(do(act_inc,S),SNew).

clog_3(S,SNew) :-
  poss(act_set(2),S),
  clog_4(do(act_set(2),S),SNew).

clog_2(S,SNew) :-
  clog_4(S,SNew).

clog_0(S,SNew) :-
  clog_1(S,SNew);
  clog_3(S,SNew).

clog_4(S,S).
```

In this case the result of the Prolog query `clog(S)` is `S = do(act_inc,s0)` or `S = do(act_set(2),s0)`, as required.

E.8 Nondeterministic choice of arguments

Consider the CLog program

```
import simple;

pick (X)
  act_set(X);
test(X == 1 || X == 2);
```

The resulting Prolog program is

```
:- ensure_loaded(simple).

clog(S) :-
  clog_0(s0,S).

clog_1(X,S,SNew) :-
  poss(act_set(X),S),
  clog_2(X,do(act_set(X),S),SNew).

clog_2(X,S,SNew) :-
  (X=1;
  X=2),
  clog_3(X,S,SNew).

clog_0(S,SNew) :-
  var(X),
  clog_1(X,S,SNew).

clog_3(X,S,SNew) :-
  clog_4(S,SNew).

clog_4(S,S).
```

Note how the compiler adds the extra variable to the arguments of all the clauses within the scope of the `pick` operation. The other is that this example will not work with numbers. The result of the Prolog query `clog(S)` is `S = do(act_set(1),s0)` or `S = do(act_set(2),s0)`.

E.9 Procedures

Consider the CLog program

```
import simple;

void proc_addTwo(X)
{
  act_inc();
  act_inc();
}

proc_addTwo(2);
```

Notice that we use the C syntax for defining procedures. The resulting Prolog program is

```
:- ensure_loaded(simple).

clog(S) :-
  clog_0(s0,S).

clog_proc_addTwo_0(X,S,SNew) :-
  poss(act_inc,S),
  clog_proc_addTwo_1(X,do(act_inc,S),SNew).

clog_proc_addTwo_1(X,S,SNew) :-
  poss(act_inc,S),
  clog_proc_addTwo_2(X,do(act_inc,S),SNew).
```

```
clog_proc_addTwo_2(X,S,S).

clog_proc_addTwo(X,S,SNew) :-
  clog_proc_addTwo_0(X,S,SNew).

clog_0(S,SNew) :-
  clog_proc_addTwo(2,S,SProc),
  clog_1(SProc,SNew).

clog_1(S,S).
```

Procedures are difficult to deal with because we must pass the correct arguments to all the sub-clauses and make sure we return to the correct point in the program. We handle the first problem by providing all sub-clauses in the scope of the procedure with all the arguments. The second problem is handled by prefixing the clause names with the procedure name and treating the body of the procedure like a new program.

In this case the result of the Prolog query `clog(S)` is `S = do(act_inc,s0)` or `S = do(act_inc,do(act_inc,s0))`, as required.

We can even handle recursive procedures. Consider the CLog program

```
import simple;

void proc_add(N)
{
  if (N > 0) {
    act_inc();
    pick (M) {
      test(M is N - 1);
      proc_add(M);
    }
  }
}

proc_add(2);
```

Notice that we use `is` because of the way Prolog handles numbers. In the absence of type information it would be preferable to always use `=` in CLog and have the compiler insert the correct Prolog depending on whether we are using numbers or not. In the absence of type information, however, this is difficult to achieve.

The resulting Prolog program is

```
:- ensure_loaded(simple).

clog(S) :-
  clog_0(s0,S).

clog_proc_add_1(N,S,SNew) :-
  poss(act_inc,S),
  clog_proc_add_2(N,do(act_inc,S),SNew).

clog_proc_add_3(N,M,S,SNew) :-
  M is N-1,
  clog_proc_add_4(N,M,S,SNew).

clog_proc_add_4(N,M,S,SNew) :-
  clog_proc_add(M,S,SProc),
  clog_proc_add_5(N,M,SProc,SNew).
```

```
clog_proc_add_2(N,S,SNew) :-
  var(M),
  clog_proc_add_3(N,M,S,SNew).

clog_proc_add_5(N,M,S,SNew) :-
  clog_proc_add_6(N,S,SNew).

clog_proc_add_0(N,S,SNew) :-
  N>0,
  clog_proc_add_1(N,S,SNew).

clog_proc_add_0(N,S,SNew) :-
  \+ (N>0),
  clog_proc_add_6(N,S,SNew).

clog_proc_add_6(N,S,S).

clog_proc_add(N,S,SNew) :-
  clog_proc_add_0(N,S,SNew).

clog_0(S,SNew) :-
  clog_proc_add(2,S,SProc),
  clog_1(SProc,SNew).

clog_1(S,S).
```

In this case the result of the Prolog query `clog(S)` is `S = do(act_inc,do(act_inc,s0))`, as required.

E.10 Miscellaneous features

In order to mimic C syntax more closely the following is a valid program:

```
void main()
{
  act_inc();
}
```

If desired, the user can drop the `void`. For many the resemblance to C may be of dubious worth. Fortunately, the lexical structure of our language can easily be modified to suit individual tastes.

Finally, we have constructs that make it easier to specify action precondition axioms, successor state axioms, and the initial state.

For example, the following program can be used in place of the `import simple` that we have been relying on up until now:

```
initially fl_counter(0);

fluent fl_counter(Y)
{
  occurrence act_inc() results in Y is X + 1 when fl_counter(X);
  occurrence act_set(X) results in Y is X;
}

action act_inc() possible when fl_counter(X) && X < 3;
action act_set(X);
```

The resulting Prolog program is

```
fl_counter(0,s0).

fl_counter(Y,do(A,S)) :-
  (
    A = act_inc,
    fl_counter(X,S),
    Y is X+1
  );
  (
    A = act_set(X),
    Y is X
  );
  (
    \+ (A = act_inc),
    \+ (A = act_set(X)),
    fl_counter(Y,S)
  ).

poss(act_inc,S) :-
  fl_counter(X,S),
  X<3.

poss(act_set(X),_).
```

Some features worth pointing out are the way of implicitly specifying defaults. In particular, notice how the `act_set(X)` action can be specified naturally, without the need for redundant qualifiers such as `when true`. We have also taken advantage of such features when specify the initial situation. The full blown version would be:

```
initially fl_counter(X) where X = 0;
```

Finally, notice that when we do not give axioms for a fluent then CLog assumes we will supply the Prolog ourselves and import it. We took advantage of this in all the previous examples in which all we stated about `fl_counter` was that it was a fluent. The user must be careful however, consider the CLog code:

```
fluent fl_counter(X)
{
}
```

In this case CLog legitimately assumes that we require it to generate the successor state axiom for us, to give:

```
fl_counter(X,do(A,S)) :-
  (
    fl_counter(X,S)
  ).
```

Appendix F

Code for Physics-based Example

The main controller is given in chapter 5. In this appendix we shall give some more of the code to fill in the details.

F.1 Procedures

In this section we give the code for the procedures that were referred to in the main controller. The purpose of our controller was to increase efficiency by avoiding considering large numbers of regions. that is the less regions we consider the faster the controller will be.

The first procedure tests the current position to see if it is adequate. If it is then the character can just stay where it is and needn't search for a new goal position.

```
proc(control_testCurrPosn(I),
  act_resetSearchRegion(I) :
  act_setEvaluator(I,currPosn) :
  act_setAcceptable(I,currPosn) :
  act_evalRegions(I)
).
```

Next we test to see if the current goal position is adequate. This takes advantage of the dynamic worlds natural temporal coherence. That is it will often be the case that the previous best goal position remains so for a while.

```
proc(control_testCurrGoalPosn(I),
  act_setEvaluator(I,goalPosn) :
  act_setAcceptable(I,goalPosn) :
  act_evalGoalRegion(I)
).
```

If the current position and the present goal position are inadequate then the character must search for a new goal position. This is done in concentric spheres around the character's current position. If at any point we find a suitable region then we stop early. At each iteration more and more regions are evaluated. The code was given back in chapter 5.

If we no suitable region exists nearby the characters will consider the swimming toward obstacles that it knows about. In effect we have given the characters the heuristic knowledge that good hiding places should

be located near obstacles.

```
proc(control_testObstacles(I),
  act_setEvaluator(I,obstacles) :
  act_setAcceptable(I,obstacles) :
  act_evalObstacles(I)
).
```

F.2 Pre-condition axioms

Most of the actions are defined to be always possible. There are, however, some important exceptions. Notably, some pre-condition axioms are referred to in some conditionals and termination conditions of some loops. For example, it is possible to pick a goal if the character has found a goal position that is at least as good as a certain threshold value.

```
poss(act_pickGoal(Index),S) :-
  fl_bestGoalPosns(Index,Worth,_,S),
  fl_acceptable(Index,Accept,S),
  Accept =< Worth.
```

In addition it is possible to expand the volume in which we search for suitable regions provided we have not exceeded some global maximum. We experimented with various values. The number of regions considered increases exponentially as we use higher values. Of course, the more regions a character considers the more likely it is to find a good one. We found `goal_constant(maxExtent,4).` to be a reasonable compromise in most cases.

```
poss(act_expandSearchRegion(Index),S) :-
  fl_searchRegion(Index,Extent,S),
  goal_constant(maxExtent,MaxExtent),
  Extent < MaxExtent.
```

Finally there is no point spending time on dead characters!

```
poss(act_updateMemory(Index),S) :-
  fl_alive(Index,true,S).
```

F.3 Successor-state axioms

The `evaluator` fluent records for each character `I` the current function that is used to evaluate the regions. It varies depending upon the status of the goal searching. For example, the criterion for evaluating obstacles wil be different to that for evaluating regions of space.

```
fl_evaluator(Index,Evaluator,do(A,S)) :-
  (
    A = act_setEvaluator(Index,Status),
    fl_evaluator(Index,OldEvaluator,S),
    fl_memory(Index,Memory,S),
    calc_setEvaluator(Index,Memory,Status,OldEvaluator,Evaluator)
  ) ;
  (
    notequal(A,act_setEvaluator(Index,_)),
    notequal(A,act_panic(Index)),
    fl_evaluator(Index,Evaluator,S)
  ).
```

Similarly, the acceptance criterion will vary depending upon the stage in the search procedure. For example, we might want to make the character more lenient when evaluating a previous goal position as

compared to searching for a new one. In particular, note that any goal will do once the character starts to panic.

```
fl_acceptable(Index,Accept,do(A,S)) :-
  (
    A = act_setAcceptable(Index,Status),
    fl_acceptable(Index,OldAccept,S),
    fl_memory(Index,Memory,S),
    calc_acceptable(Index,Memory,Status,OldAccept,Accept)
  ) ;
  (
    A = act_panic(Index),
    Accept = 0.0
  ) ;
  (
    notequal(A,act_setAcceptable(Index,_)),
    notequal(A,act_panic(Index)),
    fl_acceptable(Index,Accept,S)
  ).
```

The `fl_memory` fluent remembers the previously sensed values for the characters.

```
fl_memory(Index,Memory,do(A,S)) :-
  (
    A = act_updateMemory(Index),
    fl_sensors(Index,Sensors,S),
    fl_memory(Index,OldMemory,S),
    calc_updateMemory(OldMemory,Sensors,Memory)
  ) ;
  (
    notequal(A,act_updateMemory(Index)),
    fl_memory(Index,Memory,S)
  ).
```

The `fl_sensors` fluent stores the current sensor values. Note that the values are intervals. Currently the sensors are updated whenever the characters decide what to do next. During this time the virtual world is temporally suspended. The world is "restarted" after the `act_tock` action. The clear sensors flags resets the intervals to maximal ones thus maintaining their correctness at all times. In future we would like to try and make the intervals closer to optimal even during the periods where no sensing occurs.

```
fl_sensors(Index,Sensors,do(A,S)) :-
  (
    A = act_sense(Index),
    calc_updateSensors(Index,Sensors)
  ) ;
  (
    A = act_tock,
    calc_clearSensors(Sensors)
  ) ;
  (
    notequal(A,act_sense(Index)),
    notequal(A,act_tock),
    fl_sensors(Index,Sensors,S)
  ).
```

One of the most morbid functions of the sensors is to check whether the character has been eaten recently!

```
fl_alive(Index,Alive,do(A,S)) :-
  (
    A = act_checkAlive(Index),
    fl_sensors(Index,Sensors,S),
    calc_alive(Sensors,Alive)
  ) ;
  (
    notequal(A,act_checkAlive(Index)),
    fl_alive(Index,Alive,S)
  ).
```

The following fluent keeps track of the extent of the volume that should be searched for suitable goal positions. It is fairly self-explanatory.

```
fl_searchRegion(Index,Extent,do(A,S)) :-
  (
    A = act_resetSearchRegion(Index),
    Extent is 0
  ) ;
  (
    A = act_expandSearchRegion(Index),
    fl_searchRegion(Index,OldExtent,S),
    Extent is OldExtent + 1
  ) ;
  (
    notequal(A,act_resetSearchRegion(Index)),
    notequal(A,act_expandSearchRegion(Index)),
    fl_searchRegion(Index,Extent,S)
  ).
```

The following fluent keeps track of the best goal positions that have been found so far. Some points to note are that upon panicking the best position found so far will be used. Also, the calculation of the best

goal positions in the current search step varies depending on what the character is searching.

```
fl_bestGoalPosns(Index,Worth,GoalPosns,do(A,S)) :-
  (
    A = act_evalRegions(Index),
    fl_bestGoalPosns(Index,OldWorth,OldGoalPosns,S),
    fl_searchRegion(Index,Extent,S),
    fl_evaluator(Index,Evaluator,S),
    fl_memory(Index,Memory,S),
    calc_evalRegions(Evaluator,Index,Memory,Extent,OldWorth,Worth,
                     OldGoalPosns,GoalPosns)
  ) ;

  (
    A = act_evalObstacles(Index),
    fl_bestGoalPosns(Index,OldWorth,OldGoalPosns,S),
    fl_evaluator(Index,Evaluator,S),
    fl_memory(Index,Memory,S),
    calc_evalObstacles(Evaluator,Index,Memory,OldWorth,Worth,
                       OldGoalPosns,GoalPosns)
  ) ;
  (
    A = act_evalGoalRegion(Index),
    fl_bestGoalPosns(Index,OldWorth,OldGoalPosns,S),
    fl_evaluator(Index,Evaluator,S),
    fl_memory(Index,Memory,S),
    fl_intention(Index,Int,S),
    calc_evalGoalPosn(Evaluator,Index,Memory,Int,OldWorth,Worth,
                      OldGoalPosns,GoalPosns)
  ) ;
  (
    A = act_tock,
    Worth = 0.0,
    GoalPosns = []
  ) ;
  (
    notequal(A,act_evalRegions(Index)),
    notequal(A,act_evalObstacles(Index)),
    notequal(A,act_evalGoalRegion(Index)),
    notequal(A,act_tock),
    fl_bestGoalPosns(Index,Worth,GoalPosns,S)
  ).
```

Finally, the character's current intention is maintained. The intention is ultimately communicated to the lower level reactive behavior system. Therefore, it can be anything that the lower level reactive system knows how to perform. Here the only intention being used is to go to a goal position.

```
fl_intention(Index,Int,do(A,S)) :-
  (
    A = act_pickGoal(Index),
    fl_intention(Index,OldInt,S),
    fl_bestGoalPosns(Index,Worth,GoalPosns,S),
    calc_pickGoal(Index,Worth,GoalPosns,OldInt,Int)
  ) ;
  (
    notequal(A,act_pickGoal(Index)),
    fl_intention(Index,Int,S)
  ).
```

Bibliography

[1] G. Alefeld and Jurgen Herzberger. *Introduction to Interval Computations*. Academic Press, 1983.

[2] Alias|wavefront, `www.aw.sgi.com`. *Alias|wavefront Studio*, 1996.

[3] J. Allen, J. Hendler, and A. Tate, editors. *Readings in Planning*. Morgan Kaufmann, 1990.

[4] D. Arijon. *Grammar of the Film Language*. Communication Arts Books, Hastings House, Publishers, New York, 1976.

[5] W.W. Armstrong, M. Green, and R.Lake. Near real-time control of human figure models. In *IEEE Computer Graphics and Applications*, volume 7(6), pages 52–61, June 1987.

[6] F. Bacchus, J.Y. Halpern, and H. Levesque. Reasoning about noisy sensors in the situation calculus. In C.S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI–95)*, pages 1933–1940, Montreal, August 1995.

[7] N. I. Badler, J. O'Rourke, and G. Kaufman. Special problems in human movement simulation. In *Computer Graphics (SIGGRAPH '80 Proceedings)*, volume 14, pages 189–197, July 1980.

[8] N. I. Badler, C.B. Phillips, and D. Zeltzer. *Simulating Humans*. Oxford University Press, 1993.

[9] N. I. Badler, B. L. Webber, J. Kalita, and J. Esakov. Animation from instructions. In Norman I. Badler, Brian A. Barsky, and David Zeltzer, editors, *Making them move: mechanics, control, and animation of articulated figures*, pages 51–93. Morgan Kaufmann, 1991.

[10] N.I. Badler, B.A. Barsky, and D.Zeltzer, editors. *Making them move: mechanics, control, and animation of articulated figures*. Morgan Kaufmann, 1991.

[11] D. Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. In J. Lane, editor, *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 223–232, July 1989.

[12] D. Baraff. Curved surfaces and coherence for non-penetrating rigid body simulation. In F. Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 19–28, August 1990.

[13] D. Baraff. Coping with friction for non-penetrating rigid body simulation. In T. W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 31–40, July 1991.

[14] D. Baraff. Fast contact force computation for nonpenetrating rigid bodies. In A. Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 23–34. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[15] D. Baraff. Linear-time dynamics using lagrange multipliers. In H. Rushmeier, editor, *Proceedings of SIGGRAPH '96 (New Orleans, LA, August 4–9, 1996)*, Computer Graphics Proceedings, Annual Conference Series, pages 137–146. ACM SIGGRAPH, ACM Press, August 1996. ISBN 0-201-94800-1.

[16] D. Baraff and A. Witkin. Dynamic simulation of non-penetrating flexible bodies. In E. E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 303–308, July 1992.

[17] A. H. Barr, B. Currin, S. Gabriel, and J. F. Hughes. Smooth interpolation of orientations with angular velocity constraints using quaternions. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 313–320, July 1992.

[18] R. Barzel and A. H. Barr. A modeling system based on dynamic constraints. In J. Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 179–188, August 1988.

[19] J. Bates. The role of emotion in believable agents. *Communications of the ACM, 37, 7*, pages 122–125, 1994.

[20] M. Bauer, S. Biundo, D. Dengler, H. Feibel, J. Koehler, and G. Paul. Reasoning about plans - a progress report. In *ECAI'96 Workshop on Cross-Fertilization in Planning*, 1996.

[21] S. Bergman and A. Kaufman. BGRAF2: A real-time graphics language with modular objects and implicit dynamics. In Udo W. Pooch, editor, *Computer Graphics (SIGGRAPH '76 Proceedings)*, volume 10, pages 133–138, July 1976.

[22] A. Blake. *Canonical Expressions in Boolean Algebra*. PhD thesis, University of Chicago, 1938. Published by University of Chicago Librarries, 1938.

[23] B. M. Blumberg and T. A. Galyean. Multi-level direction of autonomous creatures for real-time environments. In R. Cook, editor, *Proceedings of SIGGRAPH '95 (Los Angeles, California)*, Computer Graphics Proceedings, Annual Conference Series, pages 47–54. ACM SIGGRAPH, ACM Press, August 1995. ISBN 0-201-84776-0.

[24] R. A. Brooks. A robot that walks: emergent behaviors from a carefully evolved network. In N.I. Badler, B.A. Barsky, and D.Zeltzer, editors, *Making Them Move: Mechanics, Control, and Animation of Articulated Figures*, pages 99–108. Morgan Kaufmann, 1991.

[25] L. S. Brotman and A. N. Netravali. Motion interpolation by optimal control. In J. Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 309–315, August 1988.

[26] A. Bruderlin and T. W. Calvert. Goal-directed, dynamic animation of human walking. In Jeffrey Lane, editor, *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 233–242, July 1989.

[27] T. W. Calvert, J. Chapman, and A. Patla. The integration of subjective and objective data in the animation of human movement. In *Computer Graphics (SIGGRAPH '80 Proceedings)*, volume 14, pages 198–203, July 1980.

[28] E. Catmull. The problems of computer-assisted animation. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12, pages 348–353, August 1978.

[29] D. T. Chen and D. Zeltzer. Pump it up: Computer animation of a biomechanically based model of muscle using the finite element method. In E. E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 89–98, July 1992.

[30] D. B. Christianson, S. E. Anderson, L. He, D. H. Salesin, D.S. Weld, and M. F. Cohen. Declarative camera control for automatic cinematography. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI–96)*, Menlo Park, CA., 1996. AAAI Press.

[31] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, December 1996. Available as CMU Tech. Report CMU-CS-96-178.

[32] M. F. Cohen. Interactive spacetime control for animation. In E. E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 293–302, July 1992.

[33] T. L. Dean and P. Wellman. *Planning and Control*. Morgan Kaufmann, 1991.

[34] J. Denavit and S. Hartenberg. A kinematic notation for lower-pair mechanisms based on matrices. In *ASME Journal of Applied Mechanics*, pages 215–221, June 1955.

[35] D. C. Dennett. *The Intentional Stance*. MIT Press, Cambridge, Mass, 1989.

[36] R.C. Dorf. *Modern control systems*. Addison-Wesley, 1992.

[37] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[38] P. Faloutsos. Physics-based animation and control of flexible characters. Technical report, University of Toronto, 1995. CSRI Technical report 326.

[39] R. Featherstone. *Robot Dynamics Algorithms*. Kluwer Academic Publishers, 1988.

[40] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics Principles and Practice*. Addison-Wesley, second edition edition, 1990.

[41] M. T. Garrett and J. D. Foley. Graphics programming using a database system with dependency declarations. *ACM Trans. on Graphics (USA)*, 1:109–128, April 1982.

[42] M.P. Gascuel. An implicit formulation for precise contact modeling between flexible solids. In J. T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 313–320, August 1993.

[43] G. De Giacomo, Y. Lespérance, and H. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI–97)*, Nagoya, Japan, August 1997.

[44] M.L. Ginsberg and D.E. Smith. Reasoning about action ii: the qualification problem. *Artificial Intelligence*, 35:311–342, 1988.

[45] M. Girard and A. A. Maciejewski. Computational modeling for the computer animation of legged figures. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 263–270, July 1985.

[46] J. A. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, Cambridge, Mass, 1995.

[47] H. Goldstein. *Classical Mechanics*. Addison-Wesley, second edition, 1980.

[48] M. Green. Using dynamics in computer animation: control and solution issues. In N. I. Badler, B. A. Barsky, and D. Zeltzer, editors, *Making Them Move: Mechanics, Control, and Animation of Articulated Figures*, pages 281–314. Morgan Kaufmann, 1991.

[49] T. R. Griffiths. *Stagecraft : The Complete Guide to Theatrical Practice*. Oxford : Phaidon, 1982.

[50] The Cognitive Robotics Group. `www.cs.utoronto.ca/~cogrobo`. Contains copies of numerous relevant papers, April 1996.

[51] R. Grzeszczuk and D. Terzopoulos. Automated learning of muscle-actuated locomotion through control abstraction. In R. Cook, editor, *Proceedings of SIGGRAPH '95 (Los Angeles, California)*, Computer Graphics Proceedings, Annual Conference Series, pages 47–54. ACM SIGGRAPH, ACM Press, August 1995. ISBN 0-201-84776-0.

[52] Radek Grzeszczuk. *Automated Learning of Muscle Locomotion Through Control Abstraction*. MSc thesis, Department of Computer Science, University of Toronto, Toronto, Canada, January 1994.

[53] H. Levesque, University of Toronto. *Personal communication*, August 1997.

[54] J. K. Hahn. Realistic animation of rigid bodies. In J. Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 299–308, August 1988.

[55] S. Hanks and D. McDermott. Temporal reasoning and default logics. Technical report, Yale University, 1985. Computer Science Research Rept. No. 430.

[56] L. He, M. F. Cohen, and D. Salesin. The virtual cinematographer: A paradigm for automatic real-time camera control and directing. In H. Rushmeier, editor, *Proceedings of SIGGRAPH '96 (New Orleans, LA, August 4–9, 1996)*, Computer Graphics Proceedings, Annual Conference Series. ACM SIGGRAPH, ACM Press, August 1996. ISBN 0-201-94800-1.

[57] E.C.R. Hehner. Boolean formalism and explanations. In *International Conference on Algebraic Methods and Software Technology*, July 1996.

[58] P. M. Isaacs and M. F. Cohen. Controlling dynamic simulation with kinematic constraints, behavior functions and inverse dynamics. In M. C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 215–224, July 1987.

[59] John Funge, `www.cs.toronto.edu/~funge/clog`. *CLog Compiler Applet*, 1997.

[60] T. Kelley. Reasoning about physical systems with the situation calculus. In *Common Sense 96, Third Symposium on Logical Formalizations of Commonsense Reasoning*, Stanford, CA, 1996.

[61] Y. Koga, K. Kondo, J. Kuffner, and J. Latombe. Planing motions with intentions. In A. Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 395–408. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[62] J. Lasseter. Principles of traditional animation applied to 3D computer animation. In M. C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 35–44, July 1987.

[63] J. C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, c1991.

[64] Y. Lee, D. Terzopoulos, and K. Waters. Realistic modeling for facial animation. In R. Cook, editor, *Proceedings of SIGGRAPH '95 (Los Angeles, California)*, Computer Graphics Proceedings, Annual Conference Series, pages 47–54. ACM SIGGRAPH, ACM Press, August 1995. ISBN 0-201-84776-0.

[65] J. Lengyel, M. Reichert, B.R. Donald, and D.P. Greenberg. Real-time robot motion planning using rasterizing computer graphics hardware. In F. Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 327–335, August 1990.

[66] Y. Lespérance, H. Levesque, F. Lin, R. Reiter D. Marcu, and R. Scherl. Foundations of a logical approach to agent programming. In *Working Notes of the IJCAI-95 Workshop on Agent Theories, Architectures, and Languages*, Montreal, August 1995.

[67] Y. Lespérance, H. Levesque, F. Lin, D. Marcu, R. Reiter, and R. Scherl. A logical approach to high-level robot programming – a progress report. In Benjamin Kuipers, editor, *Control of the Physical World by Intelligent Systems, Papers from the 1994 AAAI Fall Symposium*, pages 79–85, New Orleans, LA, November 1994.

[68] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997. Special issue on Reasoning about Action and Change.

[69] F. Lin and R. Reiter. Forget it! In Russ Greiner and Devika Subramanian, editors, *Working Notes of AAAI Fall Symposium on Relevance*, November 1994.

[70] Fangzhen Lin and Raymond Reiter. State constraints revisited. *Journal of Logic and Computation, Special Issue on Actions and Processes*, 4(5):655–678, 1994.

[71] M. C. Lin. *Efficient Collision Detection for Animation and Robotics*. PhD thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, December 1993.

[72] Z. Liu, S. J. Gortler, and M. F. Cohen. Hierarchical spacetime control. In A. Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 35–42. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[73] S. Mah, T.W. Calvert, and W. Havens. Nsail: Behavioural animation control using constraint-based reasoning. In *Proceedings of Graphics Interface '94*, pages 200–207, Banff, Alberta, Canada, May 1994. Canadian Information Processing Society.

[74] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963. Reprinted in *Semantic Information Processing* (M. Minsky ed),MIT Press, Cambridge, Mass.,1968, pages 410–417.

[75] J. McCarthy. Epistemological problems of artificial intelligence. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI–77)*, pages 1038–1044, Cambridge, MA, 1977.

[76] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, 1969.

[77] M. McKenna and D. Zeltzer. Dynamic simulation of autonomous legged locomotion. In F. Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 29–38, August 1990.

[78] C. McManis. Looking for lex and yacc for java? you don't know jack. *JavaWorld*, 1, December 1996. Written before the name change from Jack to JavaCC.

[79] D. Metaxas and D. Terzopoulos. Dynamic deformation of solid primitives with constraints. In E. E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 309–312, July 1992.

[80] G. S. P. Miller. The motion dynamics of snakes and worms. In J. Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 169–178, August 1988.

[81] M. Moore and J. Wilhelms. Collision detection and response for computer animation. In J. Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 289–298, August 1988.

[82] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[83] R. E. Moore. *Methods and Applications of Interval Analysis*. SIAM, 1979.

[84] B. A. Nayfeh. Using a cellular automata to solve mazes. *Dr. Dobb's Journal*, February 1993.

[85] A. Newell. *Unified Theories of Cognition*. Harvard University Press, 1990.

[86] V. Ng-Thow-Hing. A biomechanical musculotendon model for animating articulated objects. Master's thesis, University of Toronto, 1994.

[87] J. T. Ngo and J. Marks. Spacetime constraints revisited. In J. T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 343–350, August 1993.

[88] A. Pentland and J. Williams. Good vibrations: Modal dynamics for graphics and animation. In J. Lane, editor, *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 215–222, July 1989.

[89] J. Pesonen and E. Hyvonen. Interval approach challanges monte carlo simulation. In *Scientific Computing, Computer Arithmetic and Validated Numerics*, 1995.

[90] C. B. Phillips and N. I. Badler. Interactive behaviors for bipedal articulated figures. In T. W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 359–362, July 1991.

[91] J. Pinto and R. Reiter. Reasoning about time in the situation calculus. To appear in *Annals of Mathematics and Artificial Intelligence*, special festschrift issue in honour of Jack Minker, 1995.

[92] J. C. Platt and A. H. Barr. Constraint methods for flexible models. In J. Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 279–288, August 1988.

[93] M. H. Raibert and J. K. Hodgins. Animation of dynamic legged locomotion. In T. W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 349–358, July 1991.

[94] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honour of John McCarthy*, pages 359–380,418–420. Academic Press, 1991.

[95] C. W. Reynolds. Computer animation with scripts and actors. In *Computer Graphics (SIGGRAPH '82 Proceedings)*, volume 16, pages 289–296, July 1982.

[96] C. W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. In M. C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 25–34, July 1987.

[97] H. Rijpkema and M. Girard. Computer animation of knowledge-based human grasping. In T. W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 339–348, July 1991.

[98] R. Scherl and H. Levesque. The frame problem and knowledge-producing actions. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI–93)*, Menlo Park, CA., 1993. AAAI Press.

[99] R. J. Schilling. *Fundamentals of Robotics*. Prentice-Hall International Editions, 1990.

[100] S. Shapiro, Y. Lespérance, and H. Levesque. Goals and rationality in the situation calculus – a preliminary report. In *AAAI Fall Symposium on Rational Agency*, 1995.

[101] K. Shoemake. Animating rotation with quaternion curves. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 245–254, July 1985.

[102] K. Sims. Evolving virtual creatures. In A. Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 15–22. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[103] J. Snyder. Interval analysis for computer graphics. In E. E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 121–130, July 1992.

[104] J. M. Snyder, A. R. Woodbury, K. Fleischer, B. Currin, and A. H. Barr. Interval method for multipoint collision between time-dependent curved surfaces. In J. T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 321–334, August 1993.

[105] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, October–December 1996.

[106] S. N. Steketee and N. I. Badler. Parametric keyframe interpolation incorporating kinetic adjustment and phasing control. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 255–262, July 1985.

[107] A. J. Stewart and J. F. Cremer. Beyond keyframing: An algorithmic approach to animation. In *Proceedings of Graphics Interface '92*, pages 273–281, May 1992.

[108] J. E. Stoy. *Denotational Semantics*. MIT Press, 1977.

[109] Symbolic Dynamics Inc., Mountain View, California, USA. *SD/Fast User's Manual*, 1990.

[110] D. Terzopoulos and K. Fleischer. Modeling inelastic deformation: Viscoelasticity, plasticity, fracture. In J. Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 269–278, August 1988.

[111] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer. Elastically deformable models. In M. C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 205–214, July 1987.

[112] D. Terzopoulos, X. Tu, and R. Grzeszczuk. Artificial fishes with autonomous locomotion, perception, behavior, and learning in a simulated physical world. In *Artificial Life IV: Proc. of the Fourth International Workshop on the Synthesis and Simulation of Living System*, pages 17–27, July 1994. Cambridge, MA.

[113] X. Tu. *Artificial Animals for Computer Animation: Biomechanics, Locomotion, Perception, and Behavior*. PhD thesis, Department of Computer Science, University of Toronto, Toronto, Canada, January 1996.

[114] X. Tu and D. Terzopoulos. Artificial fishes: Physics, locomotion, perception, behavior. In A. Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 43–50. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[115] J. Tupper. *Graphing Equations with Generalized Interval Arithmetic*. MSc thesis, Department of Computer Science, University of Toronto, Toronto, Canada, January 1996.

[116] The Unicode Consortium, San Jose, CA, USA. *The Unicode Standard, Second Edition*, 1996.

[117] M. van de Panne and E. Fiume. Sensor-actuator networks. In J. T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 335–342, August 1993.

[118] M. van de Panne, E. Fiume, and Z. Vranesic. Reusable motion synthesis using state-space controllers. In F. Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 225–234, August 1990.

[119] M. van de Panne, R. Kim, and E. Fiume. Virtual wind-up toys for animation. In *Proceedings of Graphics Interface '94*, pages 208–215, Banff, Alberta, Canada, May 1994. Canadian Information Processing Society.

[120] B. Von Herzen, A. H. Barr, and H. R. Zatz. Geometric collisions for time-dependent parametric surfaces. In F. Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 39–48, August 1990.

[121] P. Wavish and M. Graham. A situated action approach to implementing characters in computer games. *Applied AI Journal*, 1995.

[122] J. Wilhelms. Using dynamic analysis for realistic animation of articulated bodies. In *IEEE Computer Graphics and Applications*, volume 7(6), pages 12–17, June 1987.

[123] A. Witkin and M. Kass. Spacetime constraints. In J. Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 159–168, August 1988.

[124] A. Witkin and W. Welch. Fast animation and control of nonrigid structures. In F. Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 243–252, August 1990.

[125] Mason Woo, Jackie Neider, and Tom Davis. *OpenGL® Programming Guide*. Addison-Wesley, 1997.

[126] Victor Ye. *A Rule-based Approach to Animating Multi-agent Environments*. PhD thesis, Department of Computer Science, University of Brighton, Brighton, England, April 1996.

[127] D. Zeltzer. Motor control techniques for figure animation. In *IEEE Computer Graphics and Applications*, volume 2, pages 289–296, 1982.

[128] R. Ziegler. *Character Animation using Transformation Based Linear Dynamics*. MSc thesis, Department of Computer Science, University of Toronto, Toronto, Canada, January 1997.

[129] S. Zilberstein and S. J. Russell. Anytime sensing, planning and action: A practical model for robot control. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, 1993.